# Emergent Solutions to High-Dimensional Multitask Reinforcement Learning

**Stephen Kelly** skelly@cs.dal.ca
Department of Computer Science, Dalhousie University, 6050 University Avenue, Halifax, NS, B3H 4R2, Canada

**Malcolm I. Heywood** mheywood@cs.dal.ca
Department of Computer Science, Dalhousie University, 6050 University Avenue, Halifax, NS, B3H 4R2, Canada

doi:10.1162/EVCO_a_00232

**Abstract**

Algorithms that learn through environmental interaction and delayed rewards, or reinforcement learning (RL), increasingly face the challenge of scaling to dynamic, high-dimensional, and partially observable environments. Significant attention is being paid to frameworks from deep learning, which scale to high-dimensional data by decomposing the task through multilayered neural networks. While effective, the representation is complex and computationally demanding. In this work, we propose a framework based on genetic programming which adaptively complexifies policies through interaction with the task. We make a direct comparison with several deep reinforcement learning frameworks in the challenging Atari video game environment as well as more traditional reinforcement learning frameworks based on a priori engineered features. Results indicate that the proposed approach matches the quality of deep learning while being a minimum of three orders of magnitude simpler with respect to model complexity. This results in real-time operation of the champion RL agent without recourse to specialized hardware support. Moreover, the approach is capable of evolving solutions to multiple game titles simultaneously with no additional computational cost. In this case, agent behaviours for an individual game as well as single agents capable of playing *all* games emerge from the same evolutionary run.

**Keywords**

Emergent modularity, cooperative coevolution, genetic programming, reinforcement learning, multitask learning.

## 1 Introduction

Reinforcement learning (RL) is an area of machine learning in which an agent develops a decision-making policy through direct interaction with a task environment. Specifically, the agent observes the environment and suggests an action based on the observation, repeating the process until a task end state is encountered. The end state provides a reward signal that characterizes quality of the policy, or the degree of success/failure. The policy's objective is therefore to select actions that maximize this long-term reward.

In real-world applications of RL, the agent is likely to observe the environment through a high-dimensional sensory interface (e.g., a video camera). This potentially implies that: (1) RL agents need to be able to assess large amounts of "low-level" information; (2) complete information about the environment is often not available from

Manuscript received: 11 July 2017; revised: 22 May 2018 and 1 June 2018; accepted: 6 June 2018.

a single observation; and (3) extended interactions and sparse rewards are common, requiring the agent to make thousands of decisions before receiving enough feedback to assess the quality of the policy. That said, the potential applications for RL are vast and diverse, from autonomous robotics (Kober and Peters, 2012) to video games (Szita, 2012), thus motivating research into RL frameworks that are general enough to be applied to a variety of environments without the use of application specific features.

Addressing dynamic, high-dimensional, and partially observable tasks in RL has recently received significant attention on account of: (1) the availability of a convenient video game emulator supporting hundreds of titles, such as the Arcade Learning Environment (ALE) (Bellemare, Naddaf et al., 2012); and, (2) human competitive results from deep learning (e.g., Mnih et al., 2015). ALE defines state, $\vec{s}(t)$, in terms of direct screen capture, while actions are limited to those of the original Atari console. Thus, learning agents interact with games via the same interface experienced by human players. In sampling 49 game titles, each designed to be interesting and challenging for human players, task environments with a wide range of properties are identified. As such, each game title requires a distinct RL policy that is capable of maximizing the score over the course of the game.

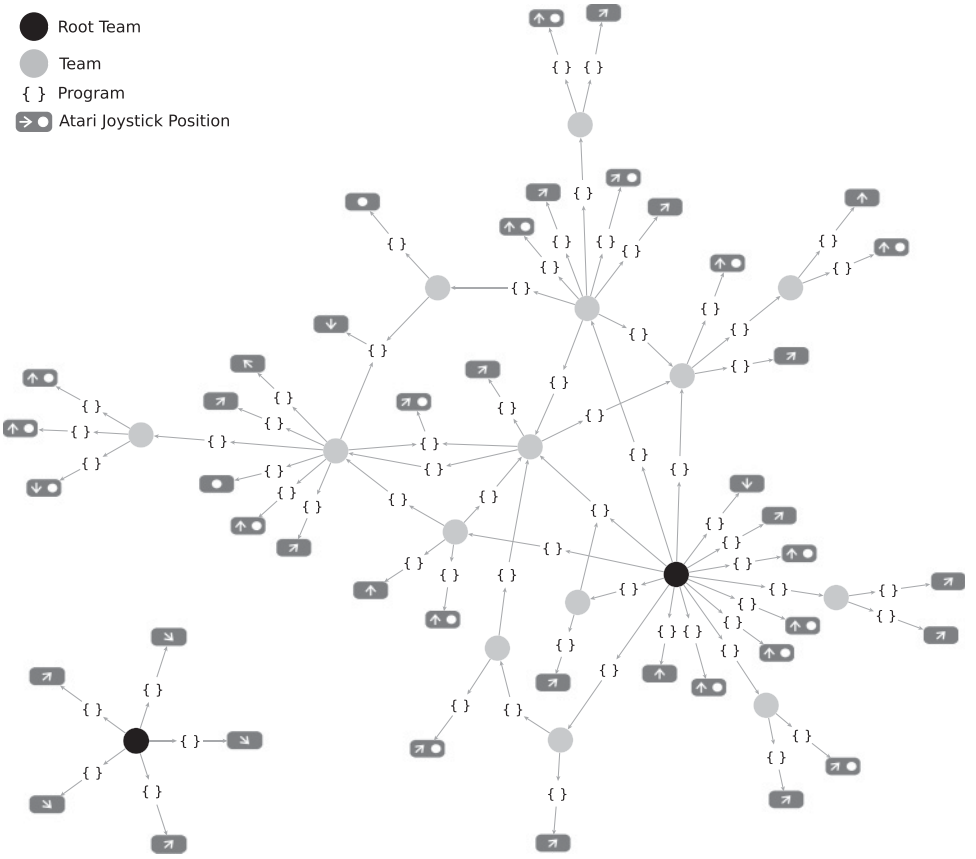
In this work, we introduce a genetic programming (GP) framework that specifically addresses challenges in scaling RL to real-world tasks while maintaining minimal model complexity. The algorithm uses emergent modularity (Nolfi, 1997) to adaptively complexify policies through interaction with the task environment. A *team* of programs represents the basic behavioural module (Lichodzijewski and Heywood, 2008b), or a mapping from state observation to an action. In sequential decision-making tasks, each program within a team defines a unique bidding behaviour (Section 3.2), such that programs cooperatively select *one action* from the team relative to the current state observation at each time step.

Evolution begins with a population of simple teams, Figure 1a, which are then further developed by adding, removing, and modifying individual programs. This work extends previous versions of an earlier (symbiotic) approach to GP teaming (Lichodzijewski and Heywood, 2011; Doucette et al., 2012; Kelly et al., 2012; Kelly and Heywood, 2014b, 2014a) to enable emergent *behavioural* modularity from a single cycle of evolution by adaptively recombining multiple teams into variably deep/wide directed graph structures, or *Tangled Program Graphs* (TPG)[1] (Figure 1b). The behaviour of each program, complement of programs per team, complement of teams per graph, and the connectivity within each graph are all emergent properties of an open-ended evolutionary process. The benefits of this approach are twofold:

1. A single graph of teams, or *policy graph*, may eventually evolve to include hundreds of teams, where each represents a simple, specialized behaviour (Figure 1b). However, mapping a state observation to an action requires traversing only *one* path through the graph from root (team) to leaf (action). Thus, the representation is capable of compartmentalizing many behaviours and recalling only those relevant to the current environmental conditions. This allows TPG to scale to complex, high-dimensional task environments while maintaining a relatively low computational cost per decision.

2. The programs in each team will collectively index a small, unique subset of the state space. As multiteam policy graphs emerge, only specific regions of the state

[1]Source code is available at https://web.cs.dal.ca/~mheywood/Code/index.html




(a) Initial Policy (b) Evolved policy

Figure 1: TPG policies. Decision making in each time step (frame) begins at the root team (black node) and follows the edge with the winning program bid (output) until an atomic action (Atari Joystick Position) is reached. The initial population contains only single-team polices (a). Multiteam graphs emerge as evolution progresses (b).

space that are important for decision making will be indexed by the graph as a whole. Thus, emergent modularity allows the policy to simultaneously decompose the task spatially *and* behaviourally, detecting important regions of the state space and optimizing the decisions made in different regions. This minimizes the requirement for a priori crafting task specific features, and lets TGP perform both feature construction and policy discovery simultaneously.

Unlike deep learning, the proposed TPG framework takes an explicitly emergent, developmental approach to policy identification. Our interest is whether we can construct policy graph topologies "bottom-up" that match the quality of deep learning solutions without the corresponding complexity. Specifically, deep learning assumes that the neural architecture is designed a priori, with the same architecture employed for each game title. Thus, deep learning always performs millions of calculations *per decision*. TPG, on the other hand, has the potential to tune policy complexity to each task

environment, or game title, requiring only $\approx 1000$ calculations per decision in the most complex case, and $\approx 100$ calculations in the simpler cases.

In short, the aim of this work is to demonstrate that much simpler solutions can be *discovered* to dynamic, high-dimensional, and partially observable environments in RL without making any prior decisions regarding model complexity. As a consequence, the computational costs typically associated with deep learning are avoided without impacting on the quality of the resulting policies, that is, the cost of training and deploying a solution is now much lower. Solutions operate in real time without any recourse to multicore or GPU hardware platforms, thus potentially simplifying the developmental/deployment overhead in posing solutions to challenging RL tasks.

Relative to our earlier work, we: (1) extend the single title comparison of 20 titles with two comparator algorithms (Kelly and Heywood, 2017a) to include all 49 Atari game titles and eight comparator algorithms (Section 5); and (2) demonstrate that multitask performance can be extended from 3 to at least 5 game titles per policy and, unlike the earlier work, does not necessitate a Pareto objective formulation (Kelly and Heywood, 2017a), just elitism (Section 7).



## 2 Background

### 2.1 Task Environment

The Arcade Learning Environment (ALE) (Bellemare, Naddaf et al., 2012) is an Atari 2600 video game emulator designed specifically to benchmark RL algorithms. The ALE allows RL agents to interact with hundreds of classic video games using the same interface as experienced by human players. That is, an RL agent is limited to interacting with the game using state, $\vec{s}(t)$, as defined by the game screen, and 18 discrete (atomic) actions, that is, the set of Atari console paddle directions including "no action," in combination with/without the fire button. Each game screen is defined by a $210 \times 160$ pixel matrix with 128 potential colours per pixel, refreshed at a frame rate of 60 Hz. In practice, the raw screen frames are preprocessed prior to being presented to an RL agent (see Section 2.2 for a summary of approaches assumed to date, and Section 4.1 for the specific approach assumed in this work).

Interestingly, important game entities often appear intermittently over sequential frames, creating visible screen flicker. This is a common technique game designers used to work around memory limitations in the original Atari hardware. However, it presents a challenge for RL because it implies that Atari game environments are partially observable. That is to say, a single frame rarely depicts the complete game state.

In addition, agents stochastically skip screen frames with probability $p = 0.25$, with the previous action being repeated on skipped frames (Bellemare, Naddaf et al., 2012; Hausknecht and Stone, 2015). This is a default setting in ALE, and aims to limit artificial agents to roughly the same reaction time as a human player as well as introducing an additional source of stochasticity. A single episode of play lasts a maximum of 18,000 frames, not including skipped frames.

### 2.2 RL under ALE Tasks

Historically, approaches to RL have relied on a priori designed task specific state representations (attributes). This changed with the introduction of the Deep Q-Network (DQN) (Mnih et al., 2015). DQN employs a deep convolutional neural network architecture to encode a representation directly from screen capture (thus a task-specific representation). A multilayer perceptron is simultaneously trained from this representation

to estimate a value function (the action selector) through Q-learning. Image preprocessing was still necessary and took the form of down sampling the original $210 \times 160$ RGB frame data to $84 \times 84$ and extracting the luminance channel. Moreover, a temporal sliding window was assumed in which the input to the first convolution layer was actually a sequence of the four most recently appearing frames. This reduced the partial observability of the task, as all the game state should now be visible.

In assuming Q-learning, DQN is an off-policy method, for which one of the most critical elements is support for replay memory. As such, performance might be sensitive to the specific content of this memory (the "memories" replayed are randomly sampled). The General Reinforcement Learning Architecture (Gorila) extended the approach of DQN with a massively parallel distributed infrastructure (100s of GPUs) to support the simultaneous development of multiple DQN learners (Nair et al., 2015). The contributions from the distributed learners periodically update a central "parameter server" that ultimately represents the solution. Gorila performed better than DQN on most game titles, but not in all cases, indicating that there are possibly still sensitivities to replay memory content.

Q-learning is also known to potentially result in action values that are excessively high. Such "overestimations" were recently shown to be associated with inaccuracies in the action values, where this is likely to be the norm during the initial stages of training (van Hasselt et al., 2016). The solution proposed by van Hasselt et al. (2016) for addressing this issue was to introduce two sets of weights, one for action selection and one for policy evaluation. This was engineered into the DQN architecture by associating the two roles with DQN's online network and target network respectively.[2] The resulting Double DQN framework improved on the original DQN results for more than half of the 49 game titles from the ALE task.

Most recently, on-policy methods (e.g., Sarsa) have appeared in which multiple independent policies are trained in parallel (Mnih et al., 2016). Each agent's experience of the environment is entirely independent (no attempt is made to enforce the centralization of memory/experience). This means that the set of RL agents collectively experience a wider range of states. The resulting evaluation under the Atari task demonstrated significant reductions to computational requirements[3] and better agent strategies. That said, in all cases, the deep learning architecture is specified a priori and subject to prior parameter tuning on a subset of game titles.

Neuro-evolution represents one of the most widely investigated techniques within the context of agent discovery for games. Hausknecht et al. (2014) performed a comparison of different neuro-evolutionary frameworks under two state representations: game title specific objects versus screen capture. Preprocessing for screen capture took the form of down sampling the original $210 \times 160$ RGB frame data to produce eight "substrates" of dimension $16 \times 21 = 336$; each substrate corresponding to one of the eight colours present in a SECAM representation (provided by the ALE). If the colour is present in the original frame data, it appears at a corresponding substrate node. Hausknecht et al. (2014) compared Hyper-NEAT, NEAT, and two simpler schemes for evolving neural networks under the suite of Atari game titles. Hyper-NEAT provides a developmental approach for describing large neural network architectures efficiently, while NEAT provides a scheme for discovering arbitrary neural topologies as well as

[2]The "online" network in DQN maintains the master copy of the MLP, whereas the target network is updated during "experience replay" (Mnih et al., 2015).

[3]A 16-core CPU as opposed to a GPU.

weight values, beginning with a single fully connected neuron. NEAT was more effective under the low-dimensional object representation, whereas Hyper-NEAT was preferable for the substrate representation.

Finally, Liang et al. (2016) revisit the design of task specific state information using a hypothesis regarding the action of the convolutional neuron in deep learning. This resulted in a state space in the order of 110 million attributes when applied to Atari screen capture, but simplified decision making to a linear model. Thus, an RL agent could be identified using the on-policy Temporal Difference method of Sarsa. In comparison to deep learning, the computational requirements for training and deployment are considerably lower, but the models produced are only as good as the ability to engineer appropriate attributes.

### 2.3 Multitask RL under ALE

The approaches reviewed in Section 2.2 assumed that a single RL policy was trained on each game title. Conversely, multitask RL (MTRL) attempts to take this further and develop a single RL agent that is able to play multiple game titles. As such, MTRL is a step towards "artificial general intelligence," and represents a much more difficult task for at least two reasons: (1) RL agents must not "forget" any of their policy for playing a previous game while learning a policy for playing a new game, and (2) during test, an RL agent must be able to distinguish between game titles without recourse to additional state information.

To date, two deep learning approaches have been proposed for this task. Parisotto et al. (2015) first learn each game title independently and then use this to train a single architecture for playing multiple titles. More recently Kirkpatrick et al. (2016) proposed a modification to Double DQN in which subsets of weights (particularly in the MLP) are associated with different tasks and subject to lower learning rates than weights not already associated with previously learned tasks. They were able to learn to play up to 6 game titles at a level comparable with the original DQN (trained on each title independently), albeit when the game titles are selected from the set of games for which DQN was known to perform well on.

## 3 Tangled Program Graphs

Modular task decomposition through teaming has been a recurring theme with genetic programming. Previous studies examined methods for combining the contribution from individual team members (Brameier and Banzhaf, 2001), enforcing island models (Imamura et al., 2003), or interchanging team-wise versus individual-wise selection (Thomason and Soule, 2007). A common limitation of such schemes was a requirement to pre-specify the number of programs appearing within a team. Moreover, even when team complement is evolved in an open ended way, it has previously been necessary to define fitness at both the level of the individual program and team (e.g., Wu and Banzhaf, 2011). Such limitations need to be addressed in order to facilitate completely open ended approaches to evolution.

### 3.1 Evolving Teams of Programs

Enabling the evolution of the number and complement of programs per team in an open manner was previously addressed in part through the use of a bidding metaphor (Lichodzijewski and Heywood, 2008a), in which case programs represent action, $a$, and context, $p$, independently. That is, each program defines the context for a single discrete action, or $a \in \{\mathcal{A}\}$ where $\mathcal{A}$ denotes the set of task specific atomic actions. Actions are

**Algorithm 1** Example program in which execution is sequential. Programs may include two-argument instructions of the form $R[x] \leftarrow R[x] \circ R[y]$ in which $\circ \in \{+, -, x, \div\}$; single-argument instructions of the form $R[x] \leftarrow \circ(R[y])$ in which $\circ \in \{cos, ln, exp\}$; and a conditional statement of the the form IF $(R[x] < R[y])$ THEN $R[x] \leftarrow -R[x]$. $R[x]$ is a reference to an internal register, while $R[y]$ may reference internal registers or state variables.

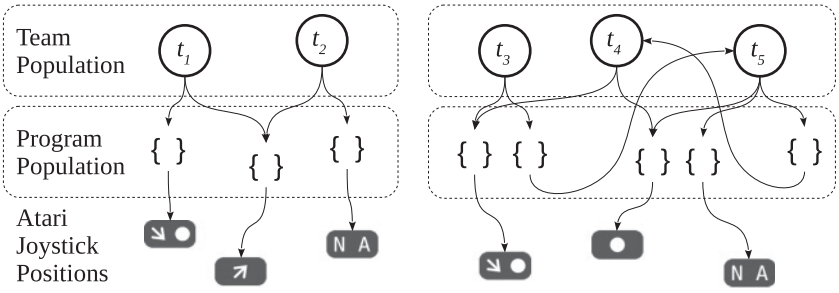
1: $R[0] \leftarrow R[0] - R[3]$
2: $R[0] \leftarrow R[0] \div R[7]$
3: $R[1] \leftarrow Log(R[0])$
4: IF $(R[0] < R[1])$ THEN $R[0] \leftarrow -R[0]$
5: RETURN $R[0]$

assigned to the program at initialization and potentially modified by variation operators during evolution. A linear program representation is assumed[4] in which a register level transfer language supports the 4 arithmetic operators, cosine, logarithmic, the exponential operation, and a conditional statement (see Algorithm 1). The linear representation facilitates skipping "intron" code, where this can potentially represent 60–70% of program instructions (Brameier and Banzhaf, 2007). Naturally, determining which of the available state variables are actually used in the program, as well as the number of instructions and their operations, are both emergent properties of the evolutionary process. After execution, register $R[0]$ represents the "bid" or "confidence" for the program's action, $a$, relative to the currently observed state, $\vec{s}(t)$. A team maps each state observation, $\vec{s}(t)$, to a single action by executing all team members (programs) relative to $\vec{s}(t)$, and then choosing the action of the highest bidder. If programs were not organized into teams, in which case all programs within the same population would compete for the right to suggest their action, it is very likely that degenerate individuals (programs that bid high for every state), would disrupt otherwise effective bidding strategies.

Adaptively building teams of programs is addressed here through the use of a symbiotic relation between a team population and a program population; hereafter "TeamGP" (Lichodzijewski and Heywood, 2008b). Each individual of the team population represents an index to some subset of the program population (see Figure 2a). Team individuals therefore assume a variable length representation in which each individual is stochastically initialized with $[2, \ldots, \omega]$ pointers to programs from the program population. The only constraint is that there must be at least two different actions indexed by the complement of programs within the same team. The same program may appear in multiple teams, but must appear in at least one team to survive between consecutive generations.

Performance (i.e., fitness) is expressed only at the level of teams, and takes the form of the task dependent objective(s). After evaluating the performance of all teams, the worst $R_{gap}$ teams are deleted from the team population. After team deletion, any program that fails to be indexed by any team must have been associated with the worst performing teams, hence is also deleted. This avoids the need to make arbitrary decisions regarding the definition of fitness at the team versus program level (which generally take the form of task specific heuristics, thus limiting the applicability of the model

[4]Any GP representation could be employed; the important innovation is that context and action are represented independently.




(a) TeamGP (b) TeamGP to TPG

Figure 2: Subplot (a) illustration of the symbiotic relation between Team and Program populations. Task fitness is only expressed at the level of a team. Each team defines a unique set of pointers to some subset of individuals from the program population. Multiple programs may have the same action, as the associated context for the action is defined by the program. Legal teams must sample at least two different actions. Subplot (b) atomic action mutated into an index to a team. There is now one less root team in the Team population.

to specific application domains). Following the deletion of the worst teams, new teams are introduced by sampling, cloning, and modifying $R_{gap}$ surviving teams. Naturally, if there is a performance benefit in smaller/larger teams and/or different program complements, this will be reflected in the surviving team–program complements (Lichodzijewski and Heywood, 2010), that is, team–program complexity is a developmental trait.

### 3.2 Evolving Graphs of Teams

Evolution begins with a program population in which program actions are limited to the task specific (atomic) actions (Figures 1a and 2a). In order to provide for the evolution of hierarchically organized code under a completely open ended process of evolution (i.e., emergent behavioural modularity), program variation operators are allowed to introduce actions that index other teams within the team population. To do so, when a program's action is modified, it has a probability ($p_{atomic}$) of referencing either a different atomic action or another team. Thus, variation operators have the ability to *incrementally* construct multiteam *policy graphs* (Figures 1b and 2b).

Each vertex in the graph is a team, while each team member, or program, represents one outgoing edge leading either to another team or an atomic action. Decision-making in a policy graph begins at the root team, where each program in the team will produce one bid relative to the current state, $\vec{s}(t)$. Graph traversal then follows the program/edge with the largest bid, repeating the bidding process for the *same* $\vec{s}(t)$ at every team/vertex along the path until an atomic action is encountered. Thus, given some state of the environment at time step $t$, the policy graph computes *one* path from root to atomic action, where only a subset of programs in the graph (i.e., those in teams along the path) require execution. Algorithm 2 details the process for evaluating the TPG individual, which is repeated at every frame, $\vec{s}(t)$, until an end-of-game state is encountered and fitness for the policy graph can be determined.

As multiteam policy graphs emerge, an increasingly tangled web of connectivity develops between the team and program populations. The number of unique solutions, or policy graphs, at any given generation is equal to the number of root nodes (i.e., teams

**Algorithm 2** Selecting an action through traversal of a policy graph. $P$ is the current program population. $\mathcal{A}$ is the set of atomic actions. $tm_i$ is the current team (initially a root node). $\vec{s}(t)$ is the state observation at time $t$. $V$ is the set of teams visited relative to state $\vec{s}(t)$ (i.e, initialize $V = \emptyset$ when $\vec{s}(t)$ is first encountered). First, all programs in $tm_i$ are executed relative to the current state $\vec{s}(t)$ (Lines 3, 4). The algorithm then considers each program in order of bid (highest to lowest, Line 6). If the program has an atomic action, the action is returned (Line 7). Otherwise, if the program's action points to a team that has not yet been visited, the procedure is called recursively (Line 9) until an action is returned (Line 7). Thus, while a policy graph may contain cycles, they are not followed during traversal. In order to ensure an atomic action is always found, team variation operators are constrained such that each team maintains at least one program that has an atomic action.

```
1: procedure SELECTACTION(tm_i, s(t), V)
2:     V = V ∪ tm_i                                  ▷ add tm_i to visited teams
3:     for all p_i ∈ tm_i do
4:         bid(p_i) = exec(p_i, s(t))                ▷ run program on s(t) and save result
5:     tm'_i = sort(tm_i)                            ▷ sort programs by bid, highest to lowest
6:     for all p_i ∈ tm'_i do
7:         if action(p_i) ∈ A then return action(p_i)        ▷ atomic action reached
8:         else if action(p_i) ∉ V then
9:             return SELECTACTION(action(p_i), s(t), V)     ▷ follow graph edge
```

that are not referenced as any program's action) in the team population. Only these root teams are candidates to have their fitness evaluated, and are subject to modification by the variation operators.

In each generation, $R_{gap}$ of the root teams are deleted and replaced by offspring of the surviving roots. The process for generating team offspring uniformly samples and clones a root team, then applies mutation-based variation operators to the cloned team which remove, add, and mutate some of its programs.

The team generation process introduces new root nodes until the number of roots in the population reaches $R_{size}$. The total number of sampling steps for generating offspring fluctuates, as root teams (along with the lower policy graph) are sometimes "subsumed" by a new team. Conversely, graphs can be separated, for example, through program action mutation, resulting in new root nodes/policies. This implies that after initialization, both team and program population size varies. Furthermore, while the number of root teams remains fixed, the number of teams that become "archived" as internal nodes (i.e., a library of reusable code) fluctuates.

Limiting evaluation, selection, and variation to root teams only has two critical benefits: (1) the cost of evaluation and the size of the search space remains low because only a fraction of the team population (root teams) represent unique policies to be evaluated and modified in each generation and (2) since only root teams are deleted, introduced, or modified, policy graphs are *incrementally* developed from the bottom up. As such, lower-level complex structures within a policy graph are protected as long as they contribute to an overall strong policy.

In summary, the teaming GP framework of Lichodzijewski and Heywood (2010) is extended to allow policy graphs to emerge, defining the inter-relation between teams. As programs composing a team typically index different subsets of the state space (i.e.,

the screen in the case of ALE), the resulting policy graph will incrementally adapt, indexing more or less of the state space *and* defining the types of decisions made in different regions. Finally, Kelly et al. (2018) provide an additional pictorial summary of the TPG algorithm.

#### 3.2.1 Neutrality Test

When variation operators introduce changes to a program, there is no guarantee that the change will: (1) result in a behavioural change, and (2) even if a behavioural change results, it will be unique relative to the current set of programs. Point 1 is still useful as it results in the potential for multiple code changes to be incrementally built up before they appear, or neutral networks (Brameier and Banzhaf, 2007). However, this can also result in wasted evaluation cycles because there is no functional difference relative to the parent. Given that fitness evaluation is expensive, we therefore test for behavioural uniqueness. Specifically, 50 of the most recent state observations are retained in a global archive, or $\vec{s}(t) \in \{t_{last} - 49, \ldots, t_{last}\}$. When a program is modified or a new program is created, its bid for each state in the archive is compared against the bid of every program in the current population. As long as all 50 bid values from the new program are not within $\tau$ of all bids from any other program in the current population, the new program is accepted. If the new program fails the test, then another instruction is mutated and the test repeated. We note that such a process has similarities with the motivation of novelty search (Lehman and Stanley, 2011), that is, a test for different outcomes. However, as this process appears at a program, there is no guarantee that this will result in any novel behaviour when it appears in a team and it is still fitness at the level of team/agents that determines survival.

## 4 Experimental Methodology

For comparative purposes, evaluation of TPG will assume the same general approach as established in the original DQN evaluation (Mnih et al., 2015). Thus, we assume the same subset of 49 Atari game titles and, post training, test the champion TPG agent under 30 test episodes initialized with a stochastically selected number of initial *no-op* actions (described in Section 5.1). This will provide us with the widest range of previous results for comparative purposes.[5] Five independent TPG runs are performed per game title, where this appears to reflect most recent practice for deep learning results.[6]

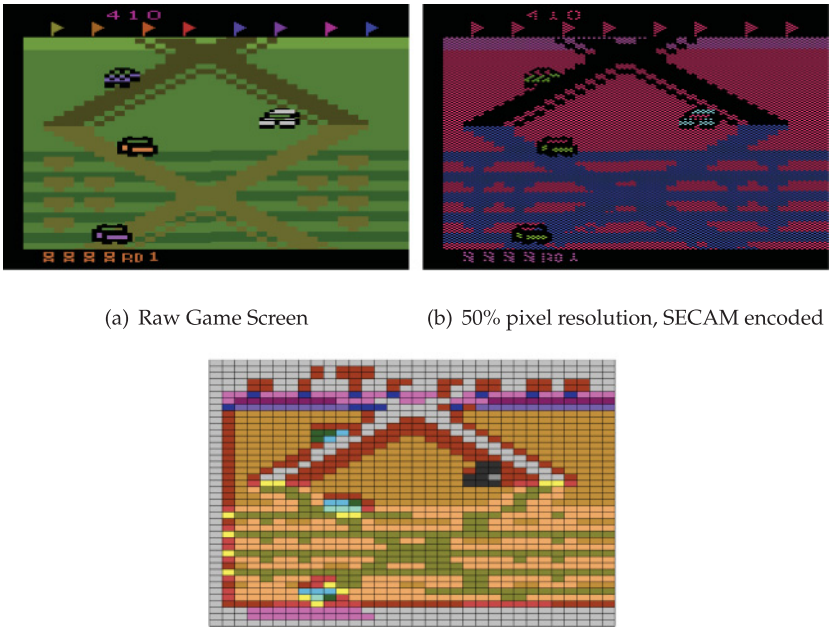
The same parameterization for TPG was used for all games (Section 4.2). The only information provided to the agents was the number of atomic actions available for each game, the preprocessed screen frame during play (Section 4.1), and the final game score. Each policy graph was evaluated in 5 game episodes per generation, up to a maximum of 10 game episodes per lifetime. Fitness for each policy graph is simply the average game score over all episodes. A single champion policy for each game was identified as that with the highest training reward at the end of evolution.

### 4.1 State Space Screen Capture

Based on the observation that the visual input has a lot of redundant information (i.e., visual game content is designed for maximizing entertainment value, as opposed to a

[5]An alternative test scenario has also appeared in which the RL agent takes over from game state identified by a human player in an attempt to introduce further diversity into RL agent start state selection (Nair et al., 2015; Mnih et al., 2016).

[6]The original DQN results only reflected a single run per title (Mnih et al., 2015).



(a) Raw Game Screen (b) 50% pixel resolution, SECAM encoded

(c) Decimal State Variables

Figure 3: Screen quantization steps, reducing the raw Atari pixel matrix (a) to 1344 decimal state variables (c) using a checkered subsampling scheme (b).

need to convey content with a minimal amount of information), we adopt a quantization approach to preprocessing. The following two-step procedure is applied to every game frame:

1. A checkered pattern mask is used to sample 50% of the pixels from the raw game screen (see Figure 3b). Each remaining pixel assumes the 8-colour SECAM encoding. The SECAM encoding is provided by ALE as an alternative to the default NTSC 128-colour format. Uniformly skipping 50% of the raw screen pixels improves the speed of feature retrieval while having minimal effect on the final representation, since important game entities are usually larger than a single pixel.
2. The frame is subdivided into a $42 \times 32$ grid.[7] Each grid tile is described by a single byte, in which each bit encodes the presence of one of eight SECAM colours within that tile. The final quantized screen representation includes each tile byte as a decimal value, so defining a state state space $\vec{s}(t)$ of $42 \times 32 = 1{,}344$ decimal features in the range of 0–255, visualized in Figure 3c for the game Up 'N Down at time step (frame) $t$.

This state representation is inspired by the Basic method defined in Bellemare, Naddaf et al. (2012). Note, however, that this method does not use a priori background detection or pairwise combinations of features.

[7] Implies that the original $210 \times 160$ screen is divided by 5.

In comparison to the DQN approach (Mnih et al., 2015; Nair et al., 2015), no attempt is made to design out the partially observable properties of game content (see discussion of Section 2.2). Moreover, the deep learning architecture's three layers of convolution filters reduce the down sampled $84 \times 84 = 7{,}056$ pixel space to a dimension of 3,136 before applying a fully connected multilayer perceptron (MLP).[8] It is the combination of convolution layer and MLP that represents the computational cost of deep learning. Naturally, this imparts a fixed computational cost of learning as the entire DQN architecture is specified a priori (Section 6.3).

In contrast, TPG evolves a decision-making agent from a 1,344 dimensional space. In common with the DQN approach, no feature extraction is performed as part of the preprocessing step, just a quantization of the original frame data. Implicit in this is an assumption that the state space is highly redundant. TPG therefore perceives the state space, $\vec{s}(t)$ (Figure 3c), as read-only memory. Each TPG program then defines a potentially unique subset of inputs from $\vec{s}(t)$ for incorporation into their decision-making process. The emergent properties of TPG are then required to develop the complexity of a solution, or policy graph, with programs organized into teams and teams into graphs. Thus, rather than assuming that all screen content contributes to decision making, the approach adopted by TPG is to adaptively subsample from the quantized image space. The specific subset of state variables sampled within each agent policy is an emergent property, discovered through interaction with the task environment alone. The implications of assuming such an explicitly emergent approach on computational cost will be revisited in Section 6.3.

### 4.2 TPG Parameterization

Deploying population-based algorithms can be expensive on account of the number of parameters and inter-relation between different parameters. In this work, no attempt has been made to optimize the parameterization (see Table 1); instead we carry over a basic parameterization from experience with evolving single teams under a supervised learning task (Lichodzijewski and Heywood, 2010).

Three basic categories of parameter are listed: Neutrality test (Section 3.2.1), Team population, and Program population (Figure 2). In the case of the Team population, the biggest parameter decisions are the population size (how many teams to simultaneously support), and how many candidate solutions to replace at each generation (*Rgap*). The parameters controlling the application of the variation operators common to earlier instances of TeamGP ($p_{md}$, $p_{ma}$, $p_{mm}$, $p_{mn}$) also assume the values used under supervised learning tasks (Lichodzijewski and Heywood, 2010). Conversely, $p_{atomic}$ represents a parameter specific to TPG, where this defines the relative chance of mutating an action to an atomic action versus a pointer to a team (Section 3.2).

Likewise, the parameters controlling properties of the Program population assume values used for TeamGP as applied to supervised learning tasks for all but $maxProgSize$. In essence this has been increased to the point where it is unlikely to be encountered during evolution. The caption of Algorithm 1 summarizes the instruction set and representation adopted for programs.

The computational limit for TPG is defined in terms of a computational resource time constraint. Thus, experiments ran on a shared cluster with a maximum runtime of 2 weeks per game title. The nature of some games allowed for >800 generations,

[8] For a tutorial on estimating the size of filters in deep learning architectures see http://cs231n.github.io/convolutional-networks/

Table 1: Parameterization of TPG.

| | |
|---|---|
| **Neutrality test (Section 3.2.1)** | |
| Number of historical samples in diversity test | 50 |
| Threshold for bid uniqueness ($\tau$) | $10^{-4}$ |
| **Team population** | |
| Number of (root) teams in initial population ($R_{size}$) | 360 |
| Number of root nodes that can be replaced per generation ($R_{gap}$) | 50% |
| Probability of deleting or adding a program ($p_{md}$, $p_{ma}$) | 0.7 |
| Max. initial team size ($\omega$) | 5 |
| Prob. of creating a new program ($p_{mm}$) | 0.2 |
| Prob. of changing a program action ($p_{mn}$) | 0.1 |
| Prob. of defining an action as a team or atomic action ($p_{atomic}$) | 0.5 |
| **Program population** | |
| Total number of registers per program (*numRegisters*) | 8 |
| Max. number of instructions a program may take (*maxProgSize*) | 96 |
| Prob. of deleting or adding an instruction within a program ($p_{delete}$, $p_{add}$) | 0.5 |
| Prob. of mutating an instruction within a program ($p_{mutate}$) | 1.0 |
| Prob. of swapping a pair of instructions within a program ($p_{swap}$) | 1.0 |

while others limited evolution to a few hundred. No attempt was made to parallelize execution within each run (i.e., the TPG code base executes as a single thread), the cluster merely enabled each run to be made simultaneously. Incidentally, the DQN results required 12–14 days per game title on a GPU computing platform (Nair et al., 2015).

## 5 Single-Task Learning

This section documents TPG's ability to build decision-making policies in the ALE from the perspective of domain-independent AI, that is, discovering policies for a variety of ALE game environments with no task-specific parameter tuning. Before presenting detailed results, we provide an overview of *training performance* for TPG on the suite of 49 ALE titles common to most benchmarking studies (Section 2.2). Figure 4 illustrates average TPG training performance (across the 5 runs per game title) as normalized relative to DQN's test score (100%) and random play (0%) (Mnih et al., 2015). The random agent simply selects actions with uniform probability at each game frame.[9] Under test conditions, TPG exceeds DQN's score in 27 games (Figure 4a), while DQN maintains the highest score in 21 games (Figure 4b). Thus, TPG and DQN are broadly comparable from a performance perspective, each matching/beating the other in a subset of game environments. Indeed, there is no statistical difference between TPG and DQN test scores over all 49 games (Section 5.1). However, TPG produces much simpler solutions in *all* cases, largely due to its emergent modular representation, which automatically scales through interaction with the task environment. That is to say, concurrent to learning a

[9]Normalized score is calculated as 100 × (TPG score − random play score)/(DQN score − random play score). Normalizing scores makes it possible to plot TPG's progress relative to multiple games together regardless of the scoring scheme in different games, and facilitates making a direct comparison with DQN.

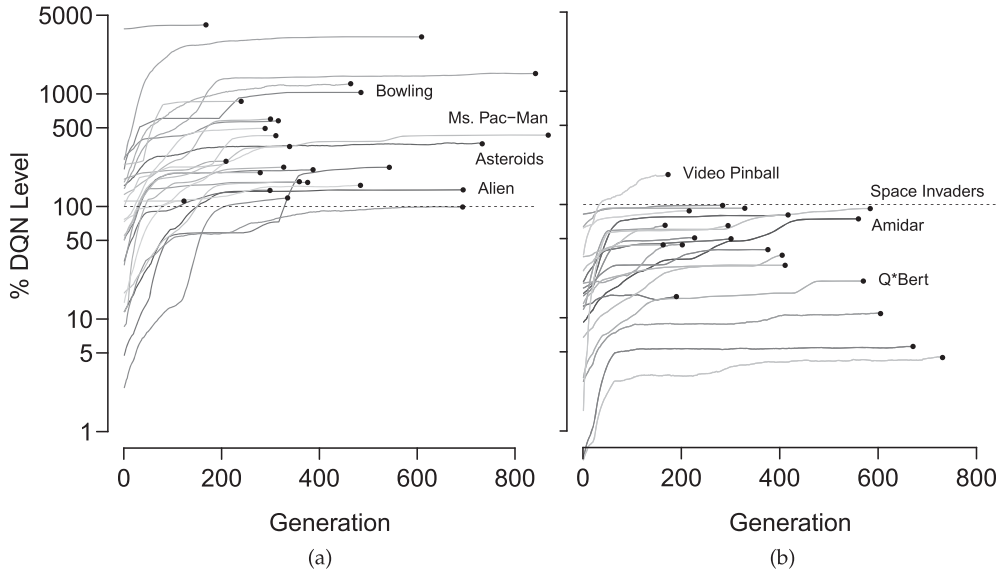



Figure 4: TPG training curves, each normalized relative to DQN's score in the same game (100%) and random play (0%): (a) shows curves for the 27 games in which TPG ultimately exceeded the level of DQN under *test* conditions and (b) shows curves for the 21 games in which TPG did not reach DQN level during test. Note that in several games TPG began with random policies (generation 1) that exceeded the level of DQN. Note that these are training scores averaged over 5 episodes in the given game title, and are thus not as robust as DQN's test score used for normalization. Also, these policies were often degenerate. For example, in Centipede, it is possible to get a score of 12,890 by selecting the "up-right-fire" action in every frame. While completely uninteresting, this strategy exceeds the test score reported for DQN (8,390) *and* the reported test score for a human professional video game tester (11,963) (Mnih et al., 2015). Regardless of their starting point, TPG policies improve throughout evolution to become more responsive and interesting. Note also that in Video Pinball, TPG exceeded DQN's score during training but not under test. The curve for Montezuma's Revenge is not pictured, a game in which neither algorithm scores any points.

strategy for gameplay, TPG explicitly answers the question of: (1) *what to index* from the state representation for each game; and (2) what components *from other candidate policies* to potentially incorporate within a larger policy. Conversely, DQN assumes a particular architecture, based on a specific deep learning–MLP combination, in which all state information *always* contributes.

### 5.1 Competency under the Atari Learning Environment

The quality of TPG policies is measured under the same test conditions as used for DQN, or the average score over 30 episodes per game title with different initial conditions and a maximum of 18,000 frames per game (Mnih et al., 2015; Nair et al., 2015). Diverse initial conditions are achieved by forcing the agent to select "no action" for the first *no-op* frames of each test game, where *no-op* $\in [0, 30]$, selected with uniform probability at

the start of each game.[10] Since some game titles derive their random seed from initial player actions, the stochastic *no-op* ensures a different seed for each test game. Stochastic frame skipping, discussed in Section 2.1, implies variation in the random seeds *and* a stochastic environment during gameplay. Both frame skipping and *no-op* are enforced in this work to ensure a stochastic environment and fair comparison to DQN. Likewise, the available actions per game are also assumed to be known.[11]

Two sets of comparator algorithm are considered:

- **Screen capture state:** construct models from game state, $\vec{s}(t)$, defined in terms of some form of screen capture input.[12] These include the original DQN deep learning results (Mnih et al., 2015), DQN as deployed through a massive distributed search (Nair et al., 2015), double DQN (van Hasselt et al., 2016), and hyper-NEAT (Hausknecht et al., 2014). While the original DQN report emphasized comparison with a human professional game tester (Mnih et al., 2015), we avoid such a comparison here primarily because the human results are not reproducible.
- **Engineered features:** define game state, $\vec{s}(t)$, in terms of features designed a priori; thus, significantly simplifying the task of finding effective policies for game play, but potentially introducing unwanted biases. Specifically, the Hyper-NEAT and NEAT results use hand crafted "Object" features specific to each game title in which different "substrates" denote the presence and location of different classes of object (see Hausknecht et al., 2014 and the discussion of Section 2.2). The Blob-PROST results assume features designed from an attempt to reverse engineer the process performed by DQN (Liang et al., 2016). The resulting state space is a vector of $\approx 110 \times 10^6$ attributes from which a linear RL agent is constructed (Sarsa). Finally, the best performing Sarsa RL agent (Conti-Sarsa) is included from the DQN study (Mnih et al., 2015) where this assumes the availability of "contingency awareness" features (Bellemare, Veness et al., 2012b).

In each case TPG based on screen capture will be compared to the set of comparator models across a common set of 49 Atari game titles. Statistical significance will be assessed using the Friedman test, where this is a nonparametric form of ANOVA (Demšar, 2006; Japkowicz and Shah, 2011). Specifically, parametric hypothesis tests assume commensurability of performance measures. This would imply that averaging results across multiple game titles makes sense. However, given that the score step size and types of property measured in each title are typically different, then averaging Null test performance across multiple titles is no longer commensurable. Conversely, the Friedman test establishes whether or not there is a pattern to the ranks. Rejecting the Null hypothesis implies that there is a pattern, and the Nemenyi post hoc test can be applied to assess the significance (Demšar, 2006; Japkowicz and Shah, 2011).

[10]Some game titles will be more affected than others. For example, titles such as Ms. Pac-Man play a song for the first ≈70 games frames while the agent's actions are ignored (thus *no-op* has no effect), while the agent takes control immediately in other game titles.

[11]The study of Liang et al. (2016) questions this assumption, but finds that better performance resulted when RL agents were constructed with the full action space.

[12]Reviewed in Section 2.2 for comparator algorithms and detailed in Section 4.1 for TPG.

In the case of RL agents derived from screen capture state information (Table 7, Appendix A), the Friedman test returns a $\chi_F^2 = 21.41$ which for the purposes of the Null hypothesis has an equivalent value from the F-distribution of $F_F = 5.89$ (Demšar, 2006). The corresponding critical value $F(\alpha = 0.01, 4, 192)$ is 3.48, hence the Null hypothesis is rejected. Applying the post hoc Nemenyi test ($\alpha = 0.05$) provides a critical difference of 0.871. Thus, relative to the best ranked algorithm (Gorila), only Hyper-NEAT is explicitly identified as outside the set of equivalently performing algorithms (or $2.63 + 0.871 < 3.87$). This conclusion is also borne out by the number of game titles for which each RL agent provides best case performance; Hyper-NEAT provides 4 best case game titles, whereas TPG, Double DQN and Gorila return at least 11 best title scores each (Table 7, Appendix A).

Repeating the process for the comparison of TPG[13] to RL agents trained under hand crafted features (Table 8), the Friedman test returns a $\chi_F^2 = 80.59$ and an equivalent value from the F-distribution of $F_F = 33.52$. The critical value is unchanged as the number of models compared and game titles is unchanged, hence the Null hypothesis is rejected. Likewise the critical difference from the post hoc Nemenyi test ($\alpha = 0.05$) is also unchanged, 0.871. This time only the performance of the Conti-Sarsa algorithm is identified as significantly worse (or $2.16 + 0.871 < 4.76$).

In summary, these results mean that despite TPG having to develop all the architectural properties of a solution, TPG is still able to provide an RL agent that performs as well as current state of the art. Conversely, DQN assumes a common prespecified deep learning topology consisting of millions of weights. Likewise, Hyper-NEAT assumes a pre-specified model complexity of ≈900,000 weights, irrespective of game title. As will become apparent in the next section, TPG is capable of evolving policy complexities that reflect the difficulty of the task.

## 6 Simplicity through Emergent Modularity

The simplest decision making entity in TPG is a single team of programs (Figure 1a), representing a standalone behaviour which maps quantized pixel state to output (action). Policies are initialized in their simplest form: as a single team with between 2 and $\omega$ programs. Each initial team will subsample a typically unique portion of the available (input) state space. Throughout evolution, search operators will develop team/program complement and may incrementally combine teams to form policy graphs. However, policies will complexify only when/if simpler solutions are outperformed. Thus, solution complexity is an evolved property driven by interaction with the task environment. By compartmentalizing decision making over multiple independent modules (teams), and incrementally combining modules into policy graphs, TPG is able to simultaneously learn which regions of the input space are important for decision making *and* discover an appropriate decision-making policy.

### 6.1 Behavioural Modularity

Emergent behavioural modularity in the development of TPG solutions can be visualized by plotting the number of teams incorporated into the champion policy graph as a function of generation (see Figure 5a). Development is nonmonotonic, and the specificity of team compliment as a function of game environment is readily apparent. For example, a game such as Asteroids may see very little in the way of increases to team complement as generations increase. Conversely, Ms. Pac-Man, which is known to be

[13] TPG still assumes screen capture state.

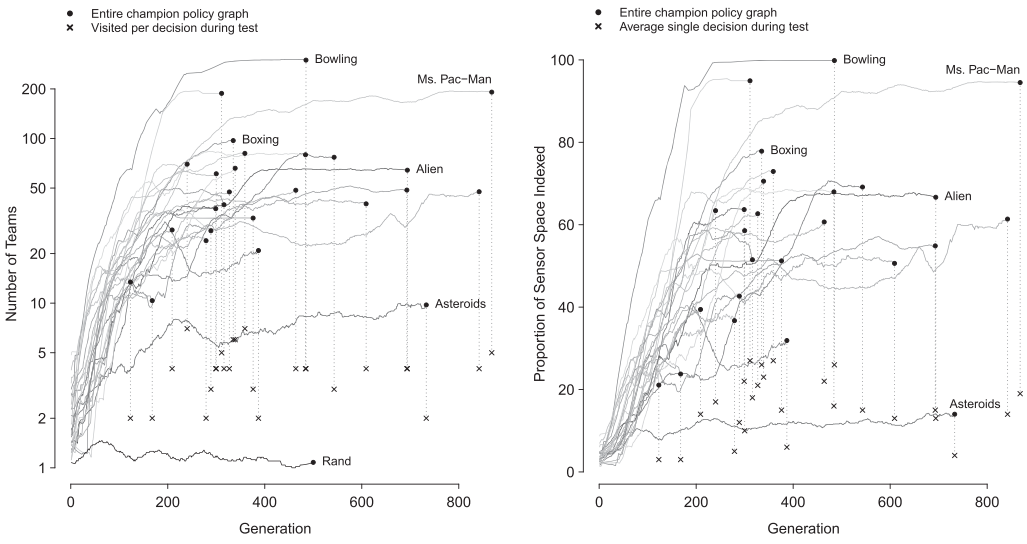
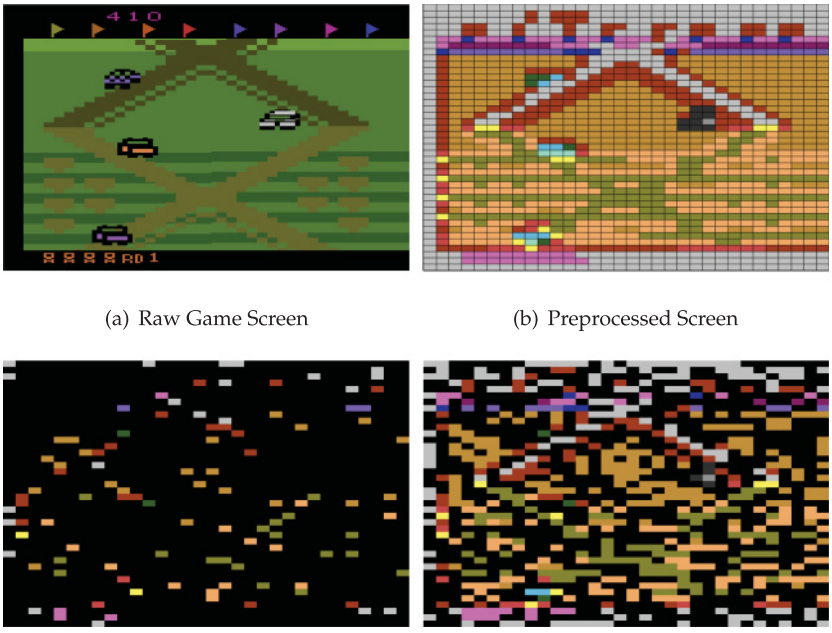



Figure 5: Emergent modularity. (a) Development of the number of teams per champion policy graph as a function of generation and game title. The run labeled "Rand" reflects the number of teams per policy when selection pressure is removed, confirming that module emergence is driven by selective pressure rather than drift or other potential biases. Black circles indicate the total number of teams in each champion policy, while $\times$ symbols indicate the average number of teams visited to make each single decision during testing. (b) Development of the proportion of input space indexed by champion policies. Black circles indicate the total proportion indexed by each champion policy, while $\times$ symbols indicate the average proportion observed to make each single decision during testing. For clarity, only the 27 game titles with TPG agent performance $\geq$DQN are depicted.

a complex task (Pepels and Winands, 2012; Schrum and Miikkulainen, 2016), saw the development of a policy graph incorporating $\approx$200 teams. Importantly, making a decision in any single time step requires following *one* path from the root team to atomic action. Thus, the cost in mapping a single game frame to an atomic action is not linearly correlated to the graph size. For example, while the number of teams in the Alien policy was $\approx$60, on average only 4 teams were visited per graph traversal during testing (see $\times$ symbols in Figure 5a). Indeed, while the total number of teams in champion TPG policy graphs ranges from 7 (Asteroids) to 300 (Bowling), the average number of teams visited per decision is typically less than 5 (Figure 5a).

### 6.2 Evolving Adapted Visual Fields

Each Atari game represents a unique graphical environment, with important events occurring in different areas of the screen, at different resolutions, and from different perspectives (e.g., global maze view versus first-person shooter). Part of the challenge with high-dimensional visual input data is determining what information is relevant to the task. Naturally, as TPG policy graphs develop, they will incrementally index more of the state space. This is likely one reason *why* they grow more in certain environments.



(a) Raw Game Screen (b) Preprocessed Screen

(c) Single Team AVF (d) Policy AVF

Figure 6: Adapted Visual Field (AVF) of champion TPG policy graph in Up 'N Down. Black regions indicate areas of the screen not indexed. (a) Shows the raw game screen. (b) Shows the preprocessed state space, where each decimal state variable (0–255) is mapped to a unique colour. (c) Shows the AVF for a single team along the active path through the policy graph at this time step, while (d) shows the AVF for the policy graph as a whole. Both AVFs exhibit patterns of sensitivity consistent with important regularities of the environment, specifically the zigzagging track.

Figure 5b plots the proportion of input space indexed by champion policy graphs throughout development, where this naturally correlates with the policy graph development shown in Figure 5a. Thus, the emergent developmental approach to model building in TPG can also be examined from the perspective of the efficiency with which information from the state space $\vec{s}(t)$ is employed. In essence, TPG policies have the capacity to develop their own Adapted Visual Fields (AVF). While the proportion of the visual field (input space) covered by a policy's AVF ranges from about 10% (Asteroids) to 100% (Bowling), the average proportion required to make each decision remains low, or less than 30% (see $\times$ symbols Figure 5b).

Figure 6 provides an illustration of the AVF as experienced by a single TPG team (c) versus the AVF for an entire champion TPG policy graph (d) in the game "Up 'N Down." This is a driving game in which the player steers a dune buggy along a track that zigzags vertically up and down the screen. The goal is to collect flags along the route and avoid hitting other vehicles. The player can smash opponent cars by jumping on them using the joystick fire button, but loses a turn upon accidentally hitting a car or jumping off the track. TPG was able to exceed the level of DQN in Up 'N Down (test games consistently ended due to the 18,000 frame limit rather than agent error) with a policy graph that indexed only 42% of the screen in total, and an average 12% of the screen per decision

(see column %SP in Table 3). The zigzagging patterns that constitute important game areas are clearly visible in the policy's AVF. In this case, the policy *learned* a simplified sensor representation well tailored to the task environment. It is also apparent that in the case of the single TPG team, the AVF does not index state information from a specific local region, but instead samples from a diverse spatial range across the entire image (Figure 6c).

In order to provide more detail, column %SP in Table 3 gives the percent of state space (screen) indexed by the policy as a whole. Maze tasks, in which the goal involves directing an avatar through a series of 2-D mazes (e.g., Bank Heist, Ms. Pac-Man, Venture) typically require near-complete screen coverage in order to navigate all regions of the maze, and relatively high-resolution is important to distinguish various game entities from maze walls. However, while the policy as a whole may index most of the screen, the modular nature of the representation implies that no more than 27% of the indexed space is considered before making each decision (Table 3, column %SP), significantly improving the runtime complexity of the policy. Furthermore, adapting the visual field implies that extensive screen coverage is only used when necessary. Indeed, in 10 of the 27 games for which TPG exceeded the score of DQN, it did so while indexing less that 50% of the screen, further minimizing the number of instructions required per decision.

In summary, while the decision-making capacity of the policy graph expands through environment-driven complexification, the modular nature of a graph representation implies that the *cost* of making each decision, as measured by the number of teams/programs which require execution, remains relatively low. Section 6.3 investigates the issue of computational cost to build solutions, and Section 6.4 will consider the cost of decision making post-training.

### 6.3 Computational Cost

The budget for model building in DQN was to assume a fixed number of decision making frames per game title (50 million). The cost of making each decision in deep learning is also fixed a priori, a function of the preprocessed image (Section 6.3) and the complexity of a multilayer perceptron (MLP). Simply put, the former provides an encoding of the original state space into a lower-dimensional space; the latter represents the decision-making mechanism.

As noted in Section 4.2, TPG runs are limited to a fixed computational time of 2 weeks per game title. However, under TPG the cost of decision making is variable as solutions do not assume a fixed topology. We can now express computational cost in terms of the cost to reach the DQN performance threshold (27 game titles), and the typical cost over the two-week period (remaining 21 game titles). Specifically, let $T$ be the generation at which a TPG run exceeds the performance of DQN. $P(t)$ denotes the number of policies in the population at generation $t$. Let $i(t)$ be the average number of instructions required by each policy to make a decision, and let $f(t)$ be the total number of frames observed over all policies at generation $t$; then the total number of operations required by TPG to discover a decision-making policy for each game is $\sum_{t=1}^{T} P(t) \times i(t) \times f(t)$. When viewed step-wise, this implies that computational cost can increase or decrease relative to the previous generation, depending on the complexity of evaluating TPG individuals (which are potentially all unique topologies).

Figure 7 plots the number of instructions required for each game over all decision-making frames observed by agents during training. Figure 7a characterizes computational cost in terms of solutions to the 27 game titles that reached the DQN performance

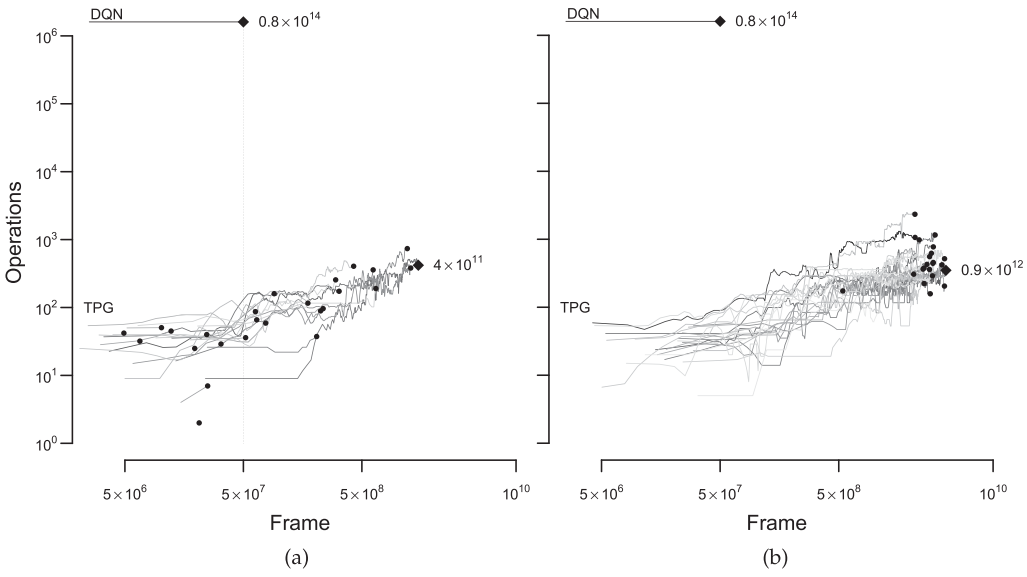



Figure 7: Number of operations per frame ($y$-axis) over all game frames observed during training ($x$-axis). (a) Shows the subset of games up to the point where TPG exceeded DQN test score. (b) Shows games for which TPG did not reach DQN test score. Black diamonds denote the most complex cases, with text indicating the cumulative number of operations required to train each algorithm up to that point. DQN's architecture is fixed a priori, thus cumulative computational cost at each frame is simply a sum over the number of operations executed up to that frame. TPG's complexity is adaptive, thus producing a unique development curve and max operations for each game title. Frame limit for DQN was 50 million ($5 \times 10^7$). Frame limit for TPG, imposed by a cluster resource time constraint of 2 weeks, is only reached in (b).

threshold, that is, the computational cost of reaching the DQN performance threshold. Conversely, Figure 7b illustrates the computational cost for games that never reached the DQN performance threshold, that is, terminated at the 2-week limit. As such, this is representative of the overall cost of model building in TPG for the ALE task given a two-week computational budget. In general, cost increases with an increasing number of (decision-making) frames, but the cost benefit of the nonmonotonic, adaptive nature of the policy development is also apparent.

It is also readily apparent that TPG typically employed more than the DQN budget for decision-making frames ($5 \times 10^7$). However, the cost of model construction is also a function of the operations per decision. For example, the parameterization adopted by DQN results in an MLP hidden layer consisting of 1,605,632 weights, or a total computational cost in the order of $0.8 \times 10^{14}$ over all 50,000,000 training frames. The total cost of TPG model building is $4 \times 10^{11}$ in the worst case (Figure 7a). Thus, the cost of the MLP step, *without* incorporating the cost of performing the deep learning convolution encoding (>3 million calculations at layer 1 for the parameterization of Mnih et al., 2015), exceeds TPG by several orders of magnitude. Moreover, this does not reflect the additional cost of performing a single weight update.

Table 2: Wall-clock time for making each decision and memory requirement.

| Method | Decisions per sec | Frames per sec | Memory |
|---|---|---|---|
| DQN | 5 | 20 | 9.8 GB |
| Blob-PROST | 56–300 | 280–1500 | 50 MB–9 GB |
| TPG | 758–2853 | 3032–11412 | 118 MB–146 MB[†] |

[†]Values for TPG reflect the memory utilized to support the *entire* population whereas only one champion agent is deployed post training, that is, tens to hundreds of kilobytes. TPG wall-clock time is measured on a 2.2-GHz Intel Xeon E5-2650 CPU.



### 6.4 Cost of Real-Time Decision Making

Table 2 summarizes the cost/resource requirement when making decisions post training, that is, the cost of deploying an agent. Liang et al. (2016) report figures for the memory and wall clock time on a 3.2-GHz Intel Core i7-4790S CPU. Computational cost for DQN is essentially static due to a fixed architecture being assumed for all games. Blob-PROST complexity is a function of the diversity of colour pallet in the game title. Apparently the 9 GB number was the worst case, with 3.7 GB representing the next largest memory requirement. It is apparent that TPG solutions are typically 2 to 3 orders of magnitude faster than DQN and an order of magnitude faster than Blob-PROST.

TPG model complexity is an evolved trait (Section 6) and only a fraction of the resulting TPG graph is ever visited per decision. Table 3 provides a characterization of this in terms of three properties of champion teams (as averaged over the 5 champions per game title, one champion per run):

- Teams (Tm)—both the average total number of teams per champion and corresponding average number of teams visited per decision.
- Instructions per decision (Ins)—the average number of instructions executed per agent decision. Note that as a linear genetic programming representation is assumed, most intron code can be readily identified and skipped for the purposes of program execution (Brameier and Banzhaf, 2007). Thus, "Ins" reflects the code actually executed.
- Proportion of visual field (%SP)—the proportion of the state space (Section 4.1) indexed by the entire TPG graph versus that actually indexed per decision. This reflects the fact that GP individuals, unlike deep learning or Blob-PROST, are never forced to explicitly index all of the state space. Instead the parts of the state space utilized per program is an emergent property (discussed in detail in Section 6.2).

It is now apparent that on average only 4 teams require evaluation per decision (values in parentheses in Tm column, Table 3). This also means that decisions are typically made on the basis of 3–27% of the available state space (values in parentheses in %SP column, Table 3). Likewise, the number of instructions executed is strongly dependent on the game title. The TPG agent for Time Pilot executed over a thousand instructions per action, whereas the TPG agent for Asteroids only executed 96. In short, rather than having to assume a fixed decision-making topology with hundreds of thousands of

Table 3: Characterizing overall TPG complexity. Tm denotes the total number of teams in champions versus the average number of teams visited per decision (value in parentheses). Ins denotes the average number of instructions executed to make each decision. %SP denotes the total proportion of the state space covered by the policy versus (value in parentheses).

| Title | Tm | Ins | %SP | Title | Tm | Ins | %SP |
|---|---|---|---|---|---|---|---|
| Alien | 67(4) | 498 | 68(13) | Amidar | 132(6) | 1066 | 87(23) |
| Assault | 37(4) | 420 | 50(14) | Asterix | 77(5) | 739 | 73(20) |
| Asteroids | 7(2) | 96 | 10(4) | Atlantis | 39(4) | 939 | 64(22) |
| Bank Heist | 94(3) | 532 | 75(15) | Battle Zone | 15(2) | 191 | 24(6) |
| Beam Rider | 115(4) | 443 | 83(13) | Bowling | 300(4) | 927 | 100(26) |
| Boxing | 102(6) | 1156 | 79(26) | Breakout | 6(3) | 158 | 8(6) |
| Centipede | 36(4) | 587 | 48(18) | C. Command | 49(4) | 464 | 54(15) |
| Crazy Climber | 150(3) | 1076 | 99(28) | Demon Attack | 19(3) | 311 | 26(8) |
| Double Dunk | 10(2) | 98 | 20(3) | Enduro | 24(3) | 381 | 37(10) |
| Fishing Derby | 33(3) | 472 | 50(15) | Freeway | 18(4) | 296 | 26(11) |
| Frostbite | 45(4) | 434 | 53(13) | Gopher | 4(2) | 156 | 8(5) |
| Gravitar | 49(4) | 499 | 62(14) | H.E.R.O. | 96(5) | 979 | 75(24) |
| Ice Hockey | 29(4) | 442 | 39(14) | James Bond | 41(4) | 973 | 59(22) |
| Kangaroo | 52(4) | 877 | 64(21) | Krull | 62(4) | 376 | 58(10) |
| Kung-Fu Master | 31(2) | 137 | 44(5) | M's Revenge | 403(2) | 722 | 100(22) |
| Ms. Pac-Man | 197(5) | 603 | 95(19) | Name This Game | 93(3) | 361 | 77(13) |
| Pong | 12(4) | 283 | 20(10) | Private Eye | 71(7) | 761 | 64(17) |
| Q*Bert | 255(8) | 2346 | 99(46) | River Raid | 7(3) | 286 | 15(9) |
| Road Runner | 86(7) | 1169 | 74(27) | Robotank | 42(2) | 252 | 47(8) |
| Seaquest | 58(4) | 579 | 60(15) | Space Invader | 68(4) | 624 | 65(17) |
| Star Gunner | 17(4) | 516 | 35(15) | Tennis | 3(2) | 71 | 5(3) |
| Time Pilot | 189(5) | 1134 | 95(27) | Tutankham | 36(2) | 464 | 58(14) |
| Up 'N Down | 28(3) | 425 | 42(12) | Venture | 77(6) | 1262 | 74(23) |
| Video Pinball | 38(3) | 399 | 55(13) | Wizard of Wor | 23(4) | 433 | 36(12) |
| Zaxxon | 81(4) | 613 | 68(16) | | | | |

parameters, TPG is capable of discovering emergent representations appropriate for each task.

## 7 Multitask Learning

Up to this point we have demonstrated the ability of TPG to build strong single-task decision-making policies, where a unique policy graph was trained from scratch for each Atari game title. This section reports on TPG's ability to learn multiple tasks simultaneously, or multitask reinforcement learning (MTRL). MTRL operates with the same state representation as single-task learning. That is, state variables consist of raw screen capture with no additional information regarding which game is currently being played. Furthermore, the full Atari action set is available to agents at all times.

When the TPG population is trained on multiple Atari games simultaneously, a single run can produce multiple champion policies, one for each game title, that match or exceed the level of play reported for DQN. In some cases, a multitask policy (i.e., a single policy graph capable of playing multiple titles) also emerges that plays *all* games

Table 4: Task groups used in multitask reinforcement learning experiments. Each group represents a set of games to be learned simultaneously (see Section 7.1).

| 3-Title Groups | Game | 5-Title Groups |
|---|---|---|
| | Alien | |
| 3.1 | Asteroids | |
| | Bank Heist | 5.1 |
| | Battle Zone | |
| 3.2 | Bowling | |
| | Centipede | |
| | Chopper Command | |
| 3.3 | Fishing Derby | 5.2 |
| | Frostbite | |
| | Kangaroo | |
| 3.4 | Krull | |
| | Kung-Fu Master | |
| | Ms. Pac-Man | 5.3 |
| 3.5 | Private Eye | |
| | Time Pilot | |

at the level of DQN. Furthermore, the training cost for TPG under MTRL is no greater than task-specific learning, and the complexity of champion multitask TPG policies is still significantly less than task-specific solutions from deep learning.

### 7.1 Task Groups

While it is possible to categorize Atari games by hand in order to support incremental learning (Braylan et al., 2015), no attempt was made here to organize game groups based on perceived similarity or multitask compatibility. Such a process would be labour intensive and potentially misleading, as each Atari game title defines its own graphical environment, colour scheme, physics, objective(s), and scoring scheme. Furthermore, joystick actions are not necessarily correlated between game titles. For example, the "down" joystick position generally causes the avatar to move vertically down the screen in maze games (e.g., Ms. Pac-Man, Alien), but might be interpreted as "pull-up" in flying games (Zaxxon), or even cause a spaceship avatar to enter hyperspace, disappearing and reappearing at a random screen location (Asteroids).

In order to investigate TPG's ability to learn multiple Atari game titles simultaneously, a variety of task groupings, that is, specific game titles to be learned simultaneously, are created from the set of games for which single-task runs of TPG performed well. Relative to the four comparison algorithms which use a screen capture state representation, TPG achieved the best reported test score in 15 of the 49 Atari game titles considered (Table 7, Appendix A). Thus, task groupings for MTRL can be created in an unbiased way by partitioning the list of 15 titles in alphabetical order. Specifically, Table 4 identifies 5 groups of 3 games each, and 3 groups of 5 games each.

### 7.2 Task Switching

As in single-task learning, each policy is evaluated in 5 episodes per generation. However, under MTRL, new policies are first evaluated in one episode under each game title in the current task group. Thereafter, the game title for each training episode is selected

with uniform probability from the set of titles in the task group. The maximum training episodes for each policy is 5 episodes under each game title. For each consecutive block of 10 generations, one title is selected with uniform probability to be the *active* title for which selective pressure is applied. Thus, while a policy may store the final score from up to 5 training episodes for each title, fitness at any given generation is the average score over up to 5 episodes in the *active title only*. Thus, selective pressure is explicitly applied only relative to a single game title. However, stochastically switching the active title at regular intervals throughout evolution implies that a policy's long-term survival is dependent on a level of competence in *all* games.

### 7.3 Elitism

There is no multiobjective fitness component in the formulation of MTRL proposed in this work. However, a simple form of elitism is used to ensure the population as a whole never entirely forgets any individual game title. As such, the single policy with the best average score in each title is protected from deletion, regardless of which title is currently active for selection. Note that this simple form of elitism does not protect multitask policies, which may not have the highest score for any single task, but *are* able to perform relatively well on multiple tasks. Failing to protect multitask policies became problematic under the methodology of our first MTRL study (Kelly and Heywood, 2017b). Thus, a simple form of multitask elitism is employed in this work. The elite multitask team is identified in each generation using the following two-step procedure:

1. Normalize each policy's mean score on each task relative to the rest of the current population. Normalized score for team $tm_i$ on task $t_j$, or $sc^n(tm_i, t_j)$, is calculated as $(sc(tm_i, t_j) - sc_{min}(t_j))/(sc_{max}(t_j) - sc_{min}(t_j))$, where $sc(tm_i, t_j)$ is the mean score for team $tm_i$ on task $t_j$ and $sc_{min,max}(t_j)$ are the population-wide min and max mean scores for task $t_j$.
2. Identify the multitask elite policy as that with the highest minimum normalized score over all tasks. Relative to all root teams in the current population, $R$, the elite multitask team is identified as $tm_i \in R \mid \forall tm_k \in R : min(sc^n(tm_i, t_{\{1..n\}}) > min(sc^n(tm_k, t_{\{1..n\}})$, where $min(sc^n(tm_i, t_{\{1..n\}})$ is the minimum normalized score for team $tm_i$ over all tasks in the game group and $n$ denotes the number of titles in the group.

Thus, in each generation, elitism identifies 1 champion team for each game title and 1 multitask champion, where elite teams are protected from deletion in that generation.

### 7.4 Parameterization

The parameterization used for TPG under multitask reinforcement learning is identical to that described in Table 1 with the exception of $R_{size}$ parameter, or the number of root teams to maintain in the population. Under MTRL, the population size was reduced to 90 (1/4 of the size used under single-task learning) in order to speed up evolution and allow more task switching cycles to occur throughout the given training period.[14] A total of 5 independent runs were conducted for each task group in Table 4. Multitask elite teams represent the champions from each run at any point during development.

[14] As under single-task experiments, the computational limit for MTRL is defined in terms of a computational time constraint. Experiments ran on a shared cluster with a maximum runtime of 1 week per run.

Table 5: Summary of multitask learning results over all task groups. MT and ST report test scores for the single best multitask (MT) and single-task (ST) policy for each game group over all 5 independent runs. Scores that match or exceed the test score reported for DQN in Mnih et al. (2015) are highlighted in grey (the MT score for Krull in group 5.3 is 90% of DQN's score, and is considered a match).

| MT | ST | Group | Game | Group | MT | ST |
|---|---|---|---|---|---|---|
| 864 | 1494.3 | | Alien | | 346.7 | 759 |
| 2176 | 2151 | 3.1 | Asteroids | | 1707 | 2181.3 |
| 1085 | 1085 | | Bank Heist | 5.1 | 724 | 724 |
| 36166 | 37100 | | Battle Zone | | 11800 | 30466.7 |
| 197 | 197 | 3.2 | Bowling | | 107 | 212 |
| 13431.2 | 22374.6 | | Centipede | | 9256.9 | 20480.2 |
| 3173.3 | 3266.7 | | Chopper Command | | 1450 | 2716.7 |
| −66.6 | −38.4 | 3.3 | Fishing Derby | 5.2 | 24.967 | 27.7 |
| 2900.7 | 4216 | | Frostbite | | 2087.3 | 4283.3 |
| 11200 | 10940 | | Kangaroo | | 11200 | 11893.3 |
| 4921.3 | 17846.7 | 3.4 | Krull | | 3644.3 | 6099.7 |
| 25600 | 42480 | | Kung-Fu Master | | 25393.3 | 34273.3 |
| 3067.7 | 3164.7 | | Ms. Pac-Man | 5.3 | 3312.3 | 3430 |
| 14665 | 14734.7 | 3.5 | Private Eye | | 4000 | 15000 |
| 7846.7 | 8193.3 | | Time Pilot | | 7270 | 8570 |

Post training, the final champions from each run are subject to the same test procedure as identified in Section 4 for each game title.

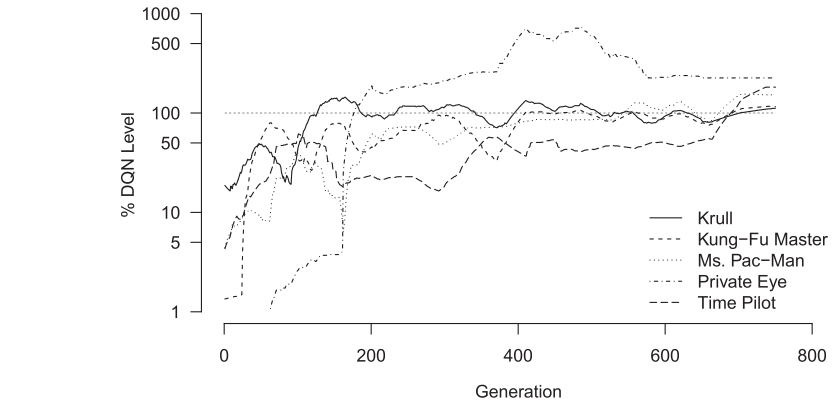
### 7.5 MTRL Performance

Figure 8 reports the MTRL training and test performance for TPG relative to game group 5.3, where all TPG scores are normalized relative to scores reported for DQN in Mnih et al. (2015). By generation ≈750, the best multitask policy is able to play all 5 game titles at the level reported for DQN.[15] Under test, the multitask champion (i.e., a single policy that plays all game titles at a high level) exceeds DQN in 4 of the 5 games, while reaching over 90% of DQN's score in the remaining title (Krull) (Figure 8b). Note that in the case of task group 5.3, only one run produced a multitask policy capable of matching DQN in all 5 tasks.

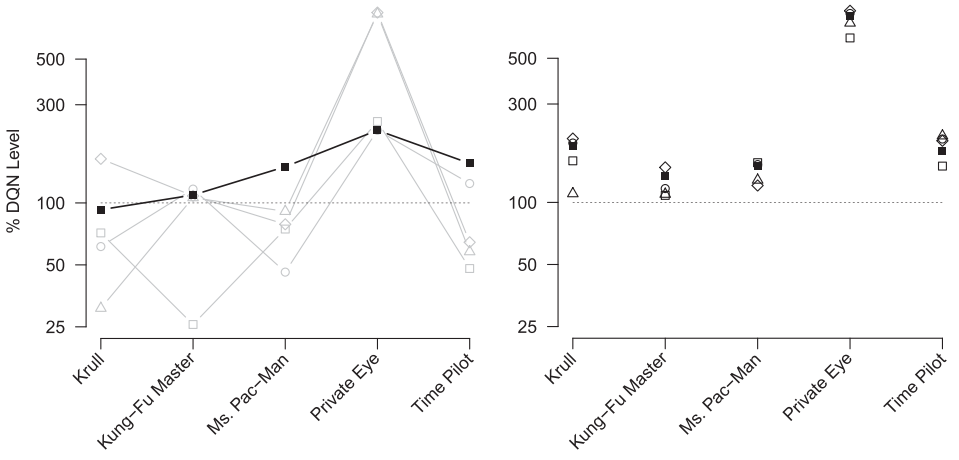
While the primary focus of MTRL is to produce multitask policies, a byproduct of the methodology employed here (i.e., task switching and elitism rather than multiobjective methods) is that each run also produces high-quality single-task policies (i.e., policies that excel at one game title). Test results for these game-specific specialists, which are simply the 5 elite single-task policies at the end of evolution, is reported in Figure 8c. While not as proficient as policies trained on a single task (Section 5.1), at least one single-task champion emerges from MTRL in task group 5.3 that matches or exceeds the score from DQN in each game title.

Table 5 provides a summary overview of test scores for the champion multi-task and single-task policy relative to each game group. Test scores that match or exceed

[15]Note that training scores reported for TPG in Figure 8a are averaged from a max of 5 episodes in each game title, and are thus not as robust as the test scores reported in Figure 8b.




(a) MTRL Train




(b) Multi-Task Test

(c) Single-Task Test

Figure 8: TPG multitask reinforcement learning results for game group 5.3. Each run identifies one elite multitask policy per generation. The training performance of this policy relative to each game title is plotted in (a), where each curve represents the mean score in each game title for the single best multitask policy over all 5 independent runs. Note that *multitask* implies that the scores reported at each generation are all from the same policy. Test scores for the final multitask champion from each of 5 runs is plotted in (b), with the single best in black. Test scores for the single-task champions from each run are plotted in (c). Note that *single-task* implies the scores are potentially all from different policies. All TPG scores are normalized relative to DQN's score in the same game (100%) and a random agent (0%). Training scores in (a) represent the policy's average score over a max of 5 episodes in each title. Test scores in (b) and (c) are the average game score over 30 test episodes in the given game title. (The line connecting points in (b) emphasizes that scores are from the same multi-task policy.) DQN scores are from Mnih et al. (2015).

that of DQN are highlighted in grey. For the 3-title groups, TPG produced multitask champions capable of playing all 3 game titles in groups 3.2, 3.4, and 3.5, while the multitask champions learned 2/3 titles in group 3.1 and only 1/3 titles in group 3.3.

For the 5-title groups, TPG produced multitask champions capable of playing all 5 titles in group 5.3, 4/5 titles in group 5.2, and 3/5 titles in group 5.1. It seems that Alien and Chopper Command are two game titles that TPG had difficulty learning under the MTRL methodology adopted here (neither multitask nor single-task policies emerged for either game title). Interestingly, while Fishing Derby was difficult to learn when grouped with Frostbite and Chopper Command (group 3.3), adding 2 additional game titles to the task switching procedure (i.e., group 5.2) seems to have been helpful to learning Fishing Derby. Note that test scores from policies developed under the MTRL methodology are generally not as high as scores achieved through single-task learning for the same game titles (Section 5.1). This is primarily due to the extra challenge of learning multiple task simultaneously. However, it is important to note that the population size for MTRL experiments was 1/4 of that used for single-task experiments and the computational budget for MTRL was half that of single-task experiments. Indeed, the MTRL results here represent a proof of concept for TPG's multitask ability rather than an exhaustive study of its full potential.

### 7.6 Modular Task Decomposition

Problem decomposition takes place at two levels in TPG: (1) program level, in which individual programs within a team each define a unique context for deploying a single action; and (2) team level, in which individual teams within a policy graph each define a unique program complement, and therefore represent a unique mapping from state observation to action. Moreover, since each program typically indexes only a small portion of the state space, the resulting mapping will be sensitive to a specific region of the state space. This section examines how modularity at the *team*-level supports the development of multitask policies.

As TPG policy graphs develop, they will subsume an increasing number of standalone decision-making modules (teams) into a hierarchical decision-making structure. Recall from Section 3.2 that only root teams are subject to modification by variation operators. Thus, teams that are subsumed as interior nodes of a policy graph undergo no modification. This property allows a policy graph to avoid (quickly) unlearning tasks that were experienced in the past under task switching but are not currently the *active* task. This represents an alternative approach to avoiding "catastrophic forgetting" (Kirkpatrick et al., 2016) during the continual, sequential learning of multiple tasks. The degree to which individual teams specialize relative to each objective experienced during evolution (that is, the game titles in a particular game group) can be characterized by looking at which teams contribute to decision making at least once during testing, relative to each game title.

Figure 9 shows the champion multitask TPG policy graph from the group 3.2 experiment. The Venn diagram indicates which teams are visited at least once while playing each game, over all test episodes. Naturally, the root team contributes to every decision (Node marked ABC in the graph, center of Venn diagram). Five teams contribute to playing both Bowling and Centipede (Node marked AB in the graph), while the rest of the teams specialize for a specific game title (Node marked A in the graph). In short, both generalist and specialist teams appear within the same policy and *collectively* define a policy capable of playing multiple game titles.

#### 7.6.1 Complexity of Multitask Policy Graphs

Table 6 reports the average number of teams, instructions, and proportion of state space contributing to each decision for the multitask champion during testing. Interestingly,

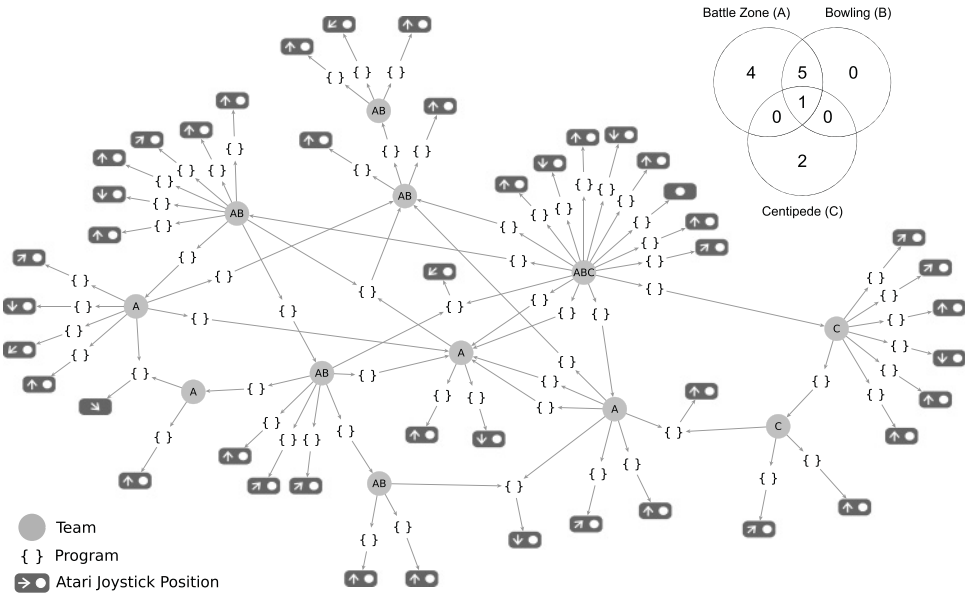



Figure 9: Champion multitask TPG policy graph from the group 3.2 experiment. Decision making in a policy graph begins at the root node (ABC) and follows *one* path through the graph until an atomic action (joystick position) is reached (see Algorithm 2). The Venn diagram indicates which teams are visited while playing each game, over all test episodes. Note that only graph nodes (teams and programs) that contributed to decision making during test are shown.

Table 6: Complexity of champion multitask policy graphs from each game group in which all tasks were covered by a single policy. The cost of making each decision is relative to the average number of teams visited per decision (Tm), average number of instructions executed per decision (Ins), and proportion of state space indexed per decision (%SP). TPG wall-clock time is measured on a 2.2-GHz Intel Xeon E5-2650 CPU.

| Group | Title | Tm | Ins | %SP | Decisions per sec |
|---|---|---|---|---|---|
| 3.2 | Battle Zone | 3 | 413 | 11 | 2687 |
| | Bowling | 4 | 499 | 15 | 2922 |
| | Centipede | 2 | 595 | 15 | 2592 |
| 3.4 | Kangaroo | 2 | 200 | 6 | 3141 |
| | Krull | 2 | 394 | 11 | 2502 |
| | Kung-Fu Master | 2 | 512 | 12 | 2551 |
| 3.5 | Ms. Pac-Man | 3 | 532 | 14 | 2070 |
| | Private Eye | 4 | 804 | 18 | 1857 |
| | Time Pilot | 5 | 869 | 19 | 1982 |
| 5.3 | Krull | 5 | 782 | 18 | 1832 |
| | Kung-Fu Master | 2 | 455 | 13 | 2342 |
| | Ms. Pac-Man | 5 | 673 | 16 | 1989 |
| | Private Eye | 3 | 481 | 13 | 2192 |
| | Time Pilot | 4 | 657 | 16 | 2306 |

even for an evolved multitask policy graph (i.e., post-training), the number of instructions executed depends on the game in play, for example, ranging from 200 in Kangaroo to 512 in Kung-Fu Master for the Group 3.4 champion. While the complexity/cost of decision making varies depending on the game in play, the average number of instructions per decision for the group 5.3 champion is 610, not significantly different from the average of 602 required by task-specific policies when playing the same games (see Table 3). Furthermore, the group 5.3 champion multitask policy averaged 1832–2342 decisions per second during testing, which is significantly faster than single-task policies from both DQN and Blob-PROST (see Table 2). Finally, as the parameterization for TPG under MTRL is identical to task-specific experiments with a significantly smaller population size (90 vs. 360), and the number of generations is similar in both cases,[16] we can conclude that the cost of development is not significantly greater under MTRL.

## 8 Conclusion

Applying RL directly to high-dimensional decision-making tasks has previously been demonstrated using both neuro-evolution and multiple deep learning architectures. To do so, neuro-evolution assumed an a priori parameterization for model complexity whereas deep learning had the entire architecture pre-specified. Moreover, evolving the deep learning architectures only optimizes the topology. The convolution operation central to providing the encoded representation remains, and it is this operation that results in the computational overhead of deep learning architectures. In this work, an entirely emergent approach to evolution, or Tangled Program Graphs, is proposed in which solution topology, state space indexing, and the types of action actually utilized are all evolved in an open ended manner.

We demonstrate that TPG is able to evolve solutions to a suite of 49 Atari game titles that generally match the quality of those discovered by deep learning at a fraction of the model complexity. To do so, TPG begins with single teams of programs and incrementally discovers a graph of interconnectivity, potentially linking hundreds of teams by the time competitive solutions are found. However, as each team can only have one action (per state), very few of the teams composing a TPG solution are evaluated in order to make each decision. This provides the basis for efficient real-time operation without recourse to specialized computing hardware. We also demonstrate a simple methodology for multitask learning with the TPG representation, in which the champion agent can play multiple games titles from direct screen capture, all at the level of deep learning, without incurring any additional training cost or solution complexity.

Future work is likely to continue investigating MTRL under increasingly high-dimensional task environments. One promising development is that TPG seems to be capable of policy discovery in VizDoom and ALE directly from the frame buffer (i.e., without the quantization procedure in Section 4.1) (e.g., Kelly et al., 2018; Smith and Heywood, 2018). That said, there are many more open issues, such as finding the relevant diversity mechanisms for tasks such as Montezuma's Revenge and providing efficient memory mechanisms that would enable agents to extend beyond the reactive models they presently assume.

[16] MTRL runs lasted 200–750 generations, which is roughly the range of generations reached for the single-task runs (see Figure 4a).

## Acknowledgments

S. Kelly gratefully acknowledges support from the Nova Scotia Graduate Scholarship program. M. Heywood gratefully acknowledges support from the NSERC Discovery program. All runs were completed on cloud computing infrastructure provided by ACENET, the regional computing consortium for universities in Atlantic Canada. The TPG code base is not in any way parallel, but in adopting ACENET the five independent runs for each of the 49 games were conducted in parallel.

## References

Bellemare, M. G., Naddaf, Y., Veness, J., and Bowling, M. (2012a). The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279.

Bellemare, M. G., Veness, J., and Bowling, M. (2012b). Investigating contingency awareness using Atari 2600 games. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pp. 864–871.

Brameier, M., and Banzhaf, W. (2001). Evolving teams of predictors with linear genetic programming. *Genetic Programming and Evolvable Machines*, 2(4):381–407.

Brameier, M., and Banzhaf, W. (2007). *Linear genetic programming*. 1st ed. New York: Springer.

Braylan, A., Hollenbeck, M., Meyerson, E., and Miikkulainen, R. (2015). Reuse of neural modules for general video game playing. Retrieved from arXiv:1512.01537.

Demšar, J. (2006). Statistical comparisons of classifiers over multiple data sets. *Journal of Machine Learning Research*, 7(1):1–30.

Doucette, J. A., Lichodzijewski, P., and Heywood, M. I. (2012). Hierarchical task decomposition through symbiosis in reinforcement learning. In *Proceedings of the ACM Genetic and Evolutionary Computation Conference*, pp. 97–104.

Hausknecht, M., Lehman, J., Miikkulainen, R., and Stone, P. (2014). A neuroevolution approach to general Atari game playing. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(4):355–366.

Hausknecht, M., and Stone, P. (2015). The impact of determinism on learning Atari 2600 games. In *AAAI Workshop on Learning for General Competency in Video Games*.

Imamura, K., Soule, T., Heckendorn, R. B., and Foster, J. A. (2003). Behavioural diversity and probabilistically optimal GP ensemble. *Genetic Programming and Evolvable Machines*, 4(3):235–254.

Japkowicz, N., and Shah, M. (2011). *Evaluating learning algorithms*. Cambridge: Cambridge University Press.

Kelly, S., and Heywood, M. I. (2014a). Genotypic versus behavioural diversity for teams of programs under the 4-v-3 keepaway soccer task. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pp. 3110–3111.

Kelly, S., and Heywood, M. I. (2014b). On diversity, teaming, and hierarchical policies: Observations from the keepaway soccer task. In *European Conference on Genetic Programming*, pp. 75–86. Lecture Notes in Computer Science, Vol. 8599.

Kelly, S., and Heywood, M. I. (2017a). Emergent tangled graph representations for Atari game playing agents. In *European Conference on Genetic Programming*, pp. 64–79. Lecture Notes in Computer Science, Vol. 10196.

Kelly, S., and Heywood, M. I. (2017b). Multi-task learning in Atari video games with emergent tangled program graphs. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 195–202.

Kelly, S., Lichodzijewski, P., and Heywood, M. I. (2012). On run time libraries and hierarchical symbiosis. In *IEEE Congress on Evolutionary Computation*, pp. 3245–3252.

Kelly, S., Smith, R., and Heywood, M. I. (2018). Emergent policy discovery for visual reinforcement learning through tangled program graphs: A tutorial. In *Genetic programming theory and practice XVI*. New York: Springer.

Kirkpatrick, J., Pascanu, R., Rabinowitz, N., Veness, J., Desjardins, G., Rusu, A. A., Milan, K., Quan, J., Ramalho, T., Grabska-Barwinska, A., Hassabis, D., Clopath, C., Kumaran, D., and Hadsell, R. (2016). Overcoming catastrophic forgetting in neural networks. Retrieved from arXiv:1612.00796.

Kober, J., and Peters, J. (2012). Reinforcement learning in robotics: A survey. In M. Wiering and M. van Otterio (Eds.), *Reinforcement learning*, pp. 579–610. New York: Springer.

Lehman, J., and Stanley, K. O. (2011). Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary Computation*, 19(2):189–223.

Liang, Y., Machado, M. C., Talvitie, E., and Bowling, M. (2016). State of the art control of Atari games using shallow reinforcement learning. In *Proceedings of the ACM International Conference on Autonomous Agents and Multiagent Systems*, pp. 485–493.

Lichodzijewski, P., and Heywood, M. I. (2008a). Coevolutionary bid-based genetic programming for problem decomposition in classification. *Genetic Programming and Evolvable Machines*, 9:331–365.

Lichodzijewski, P., and Heywood, M. I. (2008b). Managing team-based problem solving with symbiotic bid-based genetic programming. In *Proceedings of the ACM Genetic and Evolutionary Computation Conference*, pp. 863–870.

Lichodzijewski, P., and Heywood, M. I. (2010). Symbiosis, complexification and simplicity under GP. In *Proceedings of the ACM Genetic and Evolutionary Computation Conference*, pp. 853–860.

Lichodzijewski, P., and Heywood, M. I. (2011). The Rubik cube and GP temporal sequence learning: An initial study. In *Genetic programming theory and practice VIII*, chapter 3, pp. 35–54. New York: Springer.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Harley, T., Lillicrap, T. P., Silver, D., and Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. In *Proceedings of the 33rd International Conference on Machine Learning*, pp. 1928–1937.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533.

Nair, A., Srinivasan, P., Blackwell, S., Alcicek, C., Fearon, R., de Maria, A., Panneershelvam, V., Suleyman, M., Beattie, C., Petersen, S., Legg, S., Mnih, V., Kavukcuoglu, K., and Silver, D. (2015). Massively parallel methods for deep reinforcement learning. In *International Conference on Machine Learning—Deep Learning Workshop*.

Nolfi, S. (1997). Using emergent modularity to develop control systems for mobile robots. *Adaptive Behavior*, 5(3–4):343–363.

Parisotto, E., Ba, L. J., and Salakhutdinov, R. (2015). Actor-mimic: Deep multitask and transfer reinforcement learning. Retrieved from arXiv:1511.06342.

Pepels, T., and Winands, M. H. M. (2012). Enhancements for Monte-Carlo tree search in Ms Pac-Man. In *IEEE Symposium on Computational Intelligence in Games*, pp. 265–272.

Schrum, J., and Miikkulainen, R. (2016). Discovering multimodal behavior in Ms. Pac-Man through evolution of modular neural networks. *IEEE Transactions on Computational Intelligence and AI in Games*, 8(1):67–81.

Smith, R. J., and Heywood, M. I. (2018). Scaling tangled program graphs to visual reinforcement learning in Vizdoom. In *European Conference on Genetic Programming*, pp. 135–150. Lecture Notes in Computer Science, Vol. 10781.

Szita, I. (2012). Reinforcement learning in games. In M. Wiering and M. van Otterio (Eds.), *Reinforcement learning*, pp. 539–577. New York: Springer.

Thomason, R., and Soule, T. (2007). Novel ways of improving cooperation and performance in ensemble classifiers. In *Proceedings of the ACM Genetic and Evolutionary Computation Conference*, pp. 1708–1715.

van Hasselt, H., Guez, A., and Silver, D. (2016). Deep reinforcement learning with double Q-learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pp. 2094–2100.

Wu, S., and Banzhaf, W. (2011). Rethinking multilevel selection in genetic programming. In *Proceedings of the ACM Genetic and Evolutionary Computation Conference*, pp. 1403–1410.

## Appendix A: Comparator Tables

Test performance over all 49 game titles is split between two comparator groups (Section 5.1), those assuming screen capture state information (see Table 7) and those assuming hand-crafted state (see Table 8). TPG uses screen capture state in both cases.

Table 7: Average game score of best agent under test conditions for TPG along with comparator algorithms in which *screen capture* represents state information. Numbers in bold represent best score on each game title. Source information for comparitor algorithms is as follows: DQN (Mnih et al., 2015), Gorila (Nair et al., 2015), Double DQN (van Hasselt et al., 2016), and Hyper-NEAT (Hausknecht and Stone, 2015).

| Game | TPG | DQN | Gorila | Double DQN | Hyper-NEAT |
|---|---|---|---|---|---|
| Alien | **3,382.7** | 3,069.3 | 2,621.53 | 2,907.3 | 1,586 |
| Amidar | 398.4 | 739.5 | **1,189.7** | 702.1 | 184.4 |
| Assault | 2,422 | 3,359.6 | 1,450.4 | **5,022.9** | 912.6 |
| Asterix | 2,400 | 6,011.7 | 6,433.3 | **15,150** | 2,340 |
| Asteroids | **3,050.7** | 1,629.3 | 1,047.7 | 930.3 | 1,694 |
| Atlantis | 89,653 | 85,950 | **100,069** | 64,758 | 61,260 |
| Bank Heist | **1,051** | 429.7 | 609 | 728.3 | 214 |
| Battle Zone | **47,233.4** | 26,300 | 25,266.7 | 25,730 | 36,200 |
| Beam Rider | 1,320.8 | 6,845.9 | 3,302.9 | **7,654** | 1,412.8 |
| Bowling | **223.7** | 42.4 | 54 | 70.5 | 135.8 |
| Boxing | 76.5 | 71.8 | **94.9** | 81.7 | 16.4 |
| Breakout | 12.8 | 401.2 | **402.2** | 375 | 2.8 |
| Centipede | **34,731.7** | 8,309.4 | 8,432.3 | 4,139.4 | 25,275.2 |
| C. Command | **7,070** | 6,686.7 | 4,167.5 | 4,653 | 3,960 |
| Crazy Climber | 8,367 | **114,103.3** | 85,919.1 | 101,874 | 0 |
| Demon Attack | 2,920.4 | 9,711.2 | **13,693.1** | 9,711.9 | 3,590 |

Table 7: Continued.

| Game | TPG | DQN | Gorila | Double DQN | Hyper-NEAT |
|---|---|---|---|---|---|
| Double Dunk | **2** | −18.1 | −10.6 | −6.3 | **2** |
| Enduro | 125.9 | 301.8 | 114.9 | **319.5** | 93.6 |
| Fishing Derby | **49** | −0.8 | 20.2 | 20.3 | −49.8 |
| Freeway | 28.9 | 30.3 | 11.7 | **31.8** | 29 |
| Frostbite | **8,144.4** | 328.3 | 605.2 | 241.5 | 2,260 |
| Gopher | 581.4 | **8,520** | 5,279 | 8,215.4 | 364 |
| Gravitar | 786.7 | 306.7 | **1,054.6** | 170.5 | 370 |
| H.E.R.O. | 16,545.4 | 19,950.3 | 14,913.9 | **20,357** | 5,090 |
| Ice Hockey | 10 | −1.6 | −0.6 | −2.4 | **10.6** |
| James Bond | 3,120 | 576.7 | 605 | 438 | **5,660** |
| Kangaroo | **14,780** | 6,740 | 2,547.2 | 13,651 | 800 |
| Krull | **12,850.4** | 3,804.7 | 7,882 | 4,396.7 | 12,601.4 |
| Kung-Fu Master | **43,353.4** | 23,270 | 27,543.3 | 29,486 | 7,720 |
| M's Revenge | 0 | 0 | **4.3** | 0 | 0 |
| Ms. Pac-Man | **5,156** | 2,311 | 3,233.5 | 3,210 | 3,408 |
| Name This Game | 3,712 | **7,256.7** | 6,182.2 | 6,997.1 | 6,742 |
| Pong | 6 | 18.9 | 18.3 | **21** | −17.4 |
| Private Eye | **15,028.3** | 1,787.6 | 748.6 | 670.1 | 10,747.4 |
| Q*Bert | 2,245 | 10,595.8 | 10,815.6 | **14,875** | 695 |
| River Raid | 3,884.7 | 8,315.7 | 8,344.8 | **12,015** | 2,616 |
| Road Runner | 27,410 | 18,256.7 | **51,008** | 48,377 | 3,220 |
| Robotank | 22.9 | **51.6** | 36.4 | 46.7 | 43.8 |
| Seaquest | 1,368 | 5,286 | **13,169.1** | 7,995 | 716 |
| Space Invader | 1,597.2 | 1,975.5 | 1,883.4 | **3,154** | 1,251 |
| Star Gunner | 1,406.7 | 57,996.7 | 19,145 | **65,188** | 2,720 |
| Tennis | 0 | −1.6 | **10.9** | 1.7 | 0 |
| Time Pilot | **13,540** | 5,946.7 | 10,659.3 | 7,964 | 7,340 |
| Tutankham | 128 | 186.7 | **245** | 190.6 | 23.6 |
| Up 'N Down | 34,416 | 8,456.3 | 12,561.6 | 16,769.9 | **43,734** |
| Venture | 576.7 | 380 | **1,245** | 0 | 1,188 |
| Video Pinball | 37,954.4 | 42,684.1 | **157,550.2** | 70,009 | 0 |
| Wizard of Wor | 5,196.7 | 3,393.3 | **13,731.3** | 5,204 | 3,360 |
| Zaxxon | 6,233.4 | 4,976.7 | 7,129.3 | **10,182** | 3,000 |
| Avg. Rank ($R_i$) | 2.74 | 3.11 | 2.63 | 2.64 | 3.87 |

Table 8: Average game score of best agent under test conditions for TPG (screen capture) along with comparator algorithms based on *prior object/feature identification*. Numbers in bold represent best score on each game title. Source information for comparitor algorithms is as follows: Blob-PROST (Liang et al., 2016), Hyper-NEAT (Hausknecht and Stone, 2015), NEAT (Hausknecht and Stone, 2015), and Conti-Sarsa (Mnih et al., 2015).

| Game | TPG | Blob-PROST | Hyper-NEAT | NEAT | Conti-Sarsa |
|---|---|---|---|---|---|
| Alien | 3,382.7 | **4,886.6** | 2,246 | 4,320 | 103.2 |
| Amidar | 398.4 | **825.6** | 218.8 | 325.2 | 183.6 |
| Assault | 2,422 | 1,829.3 | 2,396 | **2,717.2** | 537 |
| Asterix | 2,400 | **2,965.5** | 2,550 | 1,490 | 1,332 |

Table 8: Continued.

| Game | TPG | Blob-PROST | Hyper-NEAT | NEAT | Conti-Sarsa |
|---|---|---|---|---|---|
| Asteroids | 3,050.7 | 2,229.9 | 220 | **4,144** | 89 |
| Atlantis | 89,653 | 42,937.7 | 44,200 | **126,260** | 852.9 |
| Bank Heist | 1,051 | 793.6 | **1,308** | 380 | 67.4 |
| Battle Zone | **47,233.4** | 37,850 | 37,600 | 45,000 | 16.2 |
| Beam Rider | 1,320.8 | **2,965.5** | 1,443.2 | 1,900 | 1,743 |
| Bowling | 223.7 | 91.1 | **250.4** | 231.6 | 36.4 |
| Boxing | 76.5 | **98.3** | 91.6 | 92.8 | 9.8 |
| Breakout | 12.8 | **190.3** | 40.8 | 43.6 | 6.1 |
| Centipede | **34,731.7** | 21,137 | 33,326.6 | 22,469.6 | 4,647 |
| C. Command | 7,070 | 4,898.9 | **8,120** | 4,580 | 16.9 |
| Crazy Climber | 8,367 | **81,016** | 12,840 | 25,060 | 149.8 |
| Demon Attack | 2,920.4 | 2,166 | 3,082 | **3,464** | 0 |
| Double Dunk | 2 | −4.1 | 4 | **10.8** | −16 |
| Enduro | 125.9 | **299.1** | 112.8 | 133.8 | 159.4 |
| Fishing Derby | **49** | −28.8 | −37 | −43.8 | −85.1 |
| Freeway | 28.9 | **32.6** | 29.6 | 30.8 | 19.7 |
| Frostbite | **8,144.4** | 4,534 | 2,226 | 1,452 | 180.9 |
| Gopher | 581.4 | **7,451.1** | 6,252 | 6,029 | 2,368 |
| Gravitar | 786.7 | 1,709.7 | 1,990 | **2,840** | 429 |
| H.E.R.O. | 16,545.4 | **20,273.1** | 3,638 | 3,894 | 7,295 |
| Ice Hockey | 10 | **22.8** | 9 | 3.8 | −3.2 |
| James Bond | 3,120 | 1,030.5 | **12,730** | 2,380 | 354.1 |
| Kangaroo | **14,780** | 9,492.8 | 4,880 | 12,800 | 8.8 |
| Krull | 12,850.4 | **33,263.4** | 23,890.2 | 20,337.8 | 3,341 |
| Kung-Fu Master | 43,353.4 | 51,007.6 | 47,820 | **87,340** | 29,151 |
| M's Revenge | 0 | **2,508.4** | 0 | 340 | 259 |
| Ms. Pac-Man | 5,156 | **5,917.9** | 3,830 | 4,902 | 1,227 |
| Name This Game | 3,712 | 7,787 | **8,346** | 7,084 | 2,247 |
| Pong | 6 | **20.5** | 4 | 15.2 | −17.4 |
| Private Eye | 15,028.3 | 100.3 | **15,045.2** | 1,926.4 | 86 |
| Q*Bert | 2,245 | **14,449.4** | 810 | 1,935 | 960.3 |
| River Raid | 3,884.7 | **14,583.3** | 4,736 | 4,718 | 2,650 |
| Road Runner | 27,410 | **41,828** | 14,420 | 9,600 | 89.1 |
| Robotank | 22.9 | 34.4 | **42.4** | 18 | 12.4 |
| Seaquest | 1,368 | 2,278 | **2,508** | 944 | 675.5 |
| Space Invader | **1,597.2** | 889.8 | 1,481 | 1,481 | 267.9 |
| Star Gunner | 1,406.7 | 1,651.6 | 4,160 | **9,580** | 9.4 |
| Tennis | 0 | 0 | 0.2 | **1.2** | 0 |
| Time Pilot | 13,540 | 5,429.5 | **15,640** | 14,320 | 24.9 |
| Tutankham | 128 | **217.7** | 110 | 142.4 | 98.2 |
| Up 'N Down | 34,416 | **41,257.8** | 6,818 | 10,220 | 2,449 |
| Venture | 576.7 | **1,397** | 400 | 340 | 0.6 |
| Video Pinball | 3,794.4 | 21,313 | 82,646 | **253,986** | 19,761 |
| Wizard of Wor | 5,196.7 | 5,681.2 | 3,760 | **17,700** | 36.9 |
| Zaxxon | 6233.4 | **11,721.8** | 4,680 | 6,460 | 21.4 |
| Avg. Rank ($R_i$) | 2.81 | 2.16 | 2.81 | 2.47 | 4.76 |